Verifica e validazione(12)

- **Verifica:** "ho fatto il sistema nel modo giusto", accerta che l'esecuzione delle attività di processi svolti nella fase in esame non abbia introdotto errori nel prodotto;La *software verification* ricerca la completezza e la correttezza del software e tratta ciò che lo supporta. Consente di valutare di conseguenza che il sw sia validato. La verifica è a supporto della validazione e la validazione è l'ultima cosa che faccio in un progetto. La verifica è un attività che svolgo durante **tutto lo sviluppo** fino all'ultimo istante dove farò validazione, che servirà a dire che ciò che ho fatto è la cosa giusta. La verifica va fatta per impedire che la risposta finale non sia sbagliata. Devo garantire tre cose importanti:

  - **Consistenza:** "sono ciò che vi attendevate fossi";
  - **Correttezza:** "ciò che ho conseguito è corretto rispetto alle norme";
  - **Completezza:** "tutto ciò che ho creato è tutto ciò era atteso".

  Sono tre caratteristiche di cui devo accertare l'esistenza su tutti i prodotti parziali dello sviluppo. Il verificatore impara le norme e dice che quello che è stato fornito è fatto come richiesto. Non corregge nè rifà il lavoro ma controlla solo che tutto rispetti le tre caratteristiche.

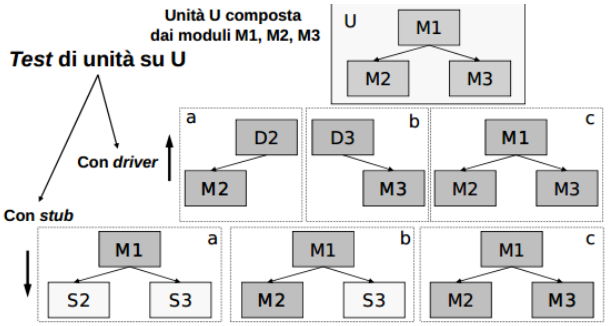
- **Validazione:** "ho fatto il sistema giusto", accerta che il prodotto realizzato sia conforme alle attese.La validazione conseguentemente è una conferma **by examination**, mostra copertura dei requisiti (utente e sw).

Per il verificatore ho due forme di **analisi**:

- **Analisi statica:** non richiede l'esecuzione del programma, studia le caratteristiche del codice sorgente (e a volte anche del codice oggetto), conformità a regole date, assenza di difetti, presenza di proprietà positive;

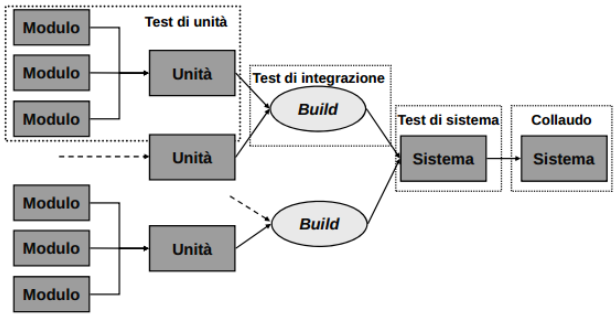
- **Analisi dinamica:** richiede l'esecuzione del programma, viene effettuata tramite **test**, usata sia nella verifica che nella validazione.

  - **Ripetibilità:** è un requisito essenziale. Dobbiamo assumere uno stato iniziale prima dell'esecuzione in quanto ha influenza sia diretta che indiretta sull'esecuzione. Il test deve essere **deterministico** ed eseguire le cose secondo un ordine noto. **Specifica di un test**;
  - **Strumenti**:
    - **Driver:** componente attiva fittizia per pilotare una parte;
    - **Stub:** componente passiva fittizia per simulare una parte;
    - **Logger:** componente non intrusiva di registrazione dei dati di esecuzione per l'analisi dei risultati. Ogni tanto deve lasciare traccia del suo esito;
  - **Unità:** può essere anche un aggregato di procedure. La più piccola unità sw che è conveniente verificare singolarmente. Un *modulo* è parte dell'unità, un *componente* integra più unità.



Con **stub** ho dei test *Top down* dalla radice alle foglie, con **driver** ho dei test *Bottom up* dalle foglie alla radice

Tipi di test



- **Test di unità:** si svolgono con il massimo grado di parallelismo, la responsabilità è dello stesso programmatore sulle unità più piccole. L'obbiettivo è quello di verificare la correttezza del codice.
- **Test di integrazione:** le componenti vengono verificate e sviluppate in parallelo, rileva errori residui nella realizzazione dei componenti, cambiamenti nelle interfacce, requisiti, integrazione con altre applicazioni non ben conosciute.
- **Test di sistema e collaudo:** dai requisiti so dire quanti test di sistema avrò. I test di sistema è un'attività interna del fornitore per accertare la copertura dei requisiti, il collaudo invece viene supervisionato dal committente.
- **Test di regressione:** è l'insieme dei test che accertano che la modifica di una parte P non causi problemi in P o in altre parte che dipendono da essa, infatti modifiche aggiunte o rimozioni non devono pregiudicare le funzionalità già verificate.

Analisi Statica

Si può applicare ai metodi di lettura che si possono suddividere in due tipi:

- **Walkthrought:** l'obbiettivo è quello di rilevare la presenza di difetti, si esegue una lettura a largo spetto senza l'assunzione di presupposti, le fasi sono: pianificazione, lettura, discussione, correzione dei difetti.
- **Inspection:** l'obbiettivo è sempre quello di rilevare difetti ma eseguendo una lettura mirata, si focalizza la ricerca su presupposti, le fasi sono: pianificazione, definizione della lista di controllo, lettura, correzione dei difetti.

Inspection è basato su errori presupposti ed è più rapido, *Walkthrought* richiede maggiore attenzione ma è più collaborativo.

Valori dell'**Analisi Statica**:

- **Funzionalità:** analisi statica come attività preliminare, liste di controllo rispetto ai relativi requisiti *(tutte e solo le funzionalità per tutti e solo i componenti necessari, compatibilità tra tutte le soluzioni adottate)*, valutazione di accuratezza;
- **Affidabilità:** dimostrabile tramite combinazione di prove, analisi statica come attività preliminare, liste di controllo rispetto ai relativi requisiti *(robustezza, capacità di ripristino e recupero da errori, adesione alle norme)*, valutazione di maturità.

- **Usabilità:** le prove sono imprescindibili, analisi statica come attività complementare, liste di controllo rispetto ai manuali d'uso *(comprensibilità, apprendibilità, adesione a norme e prescrizioni)*, questionari sottomessi agli utenti.
- **Efficenza:** le prove sono necessarie, analisi statica come attività complementare, liste di controllo rispetto alle norme di codifica, margini di miglioramento e confidenza grazie alla confidenza acquisita.
- **Manutenibità:** analisi statica come strumento ideale, liste di controllo rispetto a specifiche norme di codifica, e alla prove per accertarne, prove di stabilità.
- **Portabilità:** analisi statica come strumento ideale, liste di controllo rispetto a specifiche norme di codifica, prove come strumento complementare.

Con un software di grandi dimensioni si deve stare attenti alla sicurezza intesa come **safety**, prevenzione di condizioni di pericolo a persone o cose, e come **security**, prevenzione di intrusioni. Sw così complessi devono possedere tutte le funzionalità, specificate nei requisiti e tutte le caratteristiche non funzionali per garantire che il sistema lavori sempre come previsto.
Nessun linguaggio di programmazione garantisce a priori la completa verificabilità di ogni programma scritto con esso.

Tecniche di verifica:

- **Tracciamento:** dimostrare complettezza ed economicità della soluzione *(soddisfacimento di tutti i requisiti, nessuna funzionalità superflua, nessun componente ingiustificato)*, questo va fatto tra i requisiti software e i requisiti utente, tra sorgente e codice oggetto, tra procedure di verifica e requisiti.
  Particolari stili di codifica facilitano la verifica mediante tracciamento *assegnare singoli requisiti a singoli moduli del programma così da avere una sola procedura di prova, maggiorare l'astrazione.*
- **Revisioni:** strumento essenziale del processo di verifica, non sono automatizzabili, possono essere formali **audit** o informali **joint review**

Tipi di analisi statica:

1. **Flusso di controllo:** si accerta che il codice si esegua nella sequenza specificata, che sia ben strutturato e identifica eventuali segmenti di codice che non terminano, non sono raggiungibili;
2. **Flusso dei dati:** rileva possibili anomalie e accerta che nessun cammino di esecuzione porti ad uno stato incongruente;
3. **Flusso dell'informazione:** determinare le dipendenze in ingresso, uscita, le sole consentite sono quelle previste nella specifica;
4. **Esecuzione simbolica:** si verificano le proprietà del programma tramite la manipolazione algebrica del codice sorgente *(lo si esegue facendo sostituzioni inverse)*;
5. **Verifica formale del codice:** provare la correttezza del codice sorgente rispetto alla specifica algebrica dei requisiti, si verifica la correttezza parziale;
6. **Verifica del limite:** verificare che i dati del programma restino entro i limiti del loro tipo e dalla precisione desiderata;
7. **Uso dello stack:** si determina la massima domanda di stack richiesta da un'esecuzione in relazione con la dimensione dell'area di memoria, verificare che non ci possa essere collisione tra stack e heap;
8. **Comportamento temporale:** attenzione alle proprietà temporali richieste dalle dipendenze delle uscite dagli ingressi del programma;

9. **Interferenza:** controllare l'assenza di interferenze tra le partizioni separate del sistema;
10. **Codice oggetto:** assicurarsi che il codice oggetto sia una traduzione esatta del codice sorgente, che non ci siano errori o omissioni introdotte dal compilatore.

La verifica solo retrospettiva *(a valle dello sviluppo)* è spesso inadeguata.

Eseguire cicli revisione, verifica dopo ogni rilevazione d'errore è troppo oneroso, meglio effettuare analisi statiche durante la codifica. Si deve cercare di avere delle linee guida per la codifica del codice come prediligere o proibire l'uso di particolari costrutti, *separare le interfacce dall'implementazione, massimizzare l'implementazione, massimizzare l'incapsulazione, usare tipi specializzati per specificare i dati.*

Analisi Dinamica

Analisi dinamica=test, la prova che consiste nella verifica dinamica del comportamento del programma, queste prove si faranno su un insieme finito di casi selezionati nel dominio delle esecuzioni possibili.

Dentro la classificazione ci deve essere:

- Terminologia, Fault,Error,Failure;
  - **Failure:** quando il comportamento del sistema devia da quello che ci si aspetta;
  - **Error:** possono essere meccanici, di algoritmo o concettuali e causano guasti *Fault* terminali;
  - **Fault:** sono quelli che fanno si che esista l'errore.
- Fondamenti teorici;
- Oggetti delle prove;
- Obbiettivi delle prove: installazione nell'ambiente di prova, accettazione, collaudo;
- Vincoli di progetto: definizione del processo dei prodotti e del personale addetto alle prove, stima e controllo dei costi;
- Attività di prova: pianificazione e specifica dei casi di prova, sviluppo ambiente di prova.

La strategia di prova va bene bilanciata tra la *quantità minima* di casi di prova sufficienti a fornire certezza sulla qualità di prodotto e la *quantità massima* di sforzo, tempo e risorse per il completamento della verifica. Nel PdQ devono essere specificate quali e quante prove. Una singola prova non basta, deve infatti essere ripetibile.
3 teoremi che parlano della verifica del software:

- **Teorema di Howden:** Non esiste nessun algoritmo che dato un programma P qualsiasi generi un test ideale (da un criterio di affidabilità e valido);
- **Tesi di Dijksta:** I test su di un programma possono rilevare la presenza di malfunzionamenti, ma non dimostrarne l'assenza;

Una valida strategia di assemblaggio delle parti è quella incrementale, in quanto i difetti rilevati da un test molto probabilmente saranno relativi all'ultima parte integrata, cercare anche di assemblare prima i produttori dei consumatori così da avere un flusso di controllo e dei dati corretti per i secondi.
Una volta adottata questa strategia i metodi per svolgerla sono due:

- **Bottom-up:** si sviluppano prima le parti con maggiore utilità e minor dipendenza funzionale. Questo tipo di approccio riduce il numero di *stub* necessari al test ma le funzionalità di alto livello sono ritardate;

- **Top-down:** si sviluppano prima le parti più esterne, questa strategia comporta l'uso di molti *stub* ma integra le funzionalità a partire dall'alto.

TEST:

- **Test di unità:** un'unità software è composta da uno o più moduli, 2/3 dei difetti vengono identificati da questi test. Per ogni test si definiscono strategia, oggetto da testare, risorse necessarie e piano di esecuzione. Si ha **Statement Coverage** quando i test effettuati sull'unità sono sufficienti ad eseguire tutte le linee di comando di ciascun dei moduli, si ha **Branch Coverage** quando ciascun ramo del flusso di controllo viene attraversato almeno una volta.
  Test funzionale: (black-box) da solo non può accertare correttezza e completezza della logica interna dell'unità, ciascun insieme di dati in ingresso producono un dato comportamento funzionale che costituisce un caso di prova.
  Test strutturale: (white-box) verifica la logica interna del codice dell'unità cercando massima copertura, le prove devono essere progettate per attivare ogni cammino di esecuzione all'interno del modulo, ciascun insieme di dati che attiva un percorso costituisce un caso di prova.

- **Test di integrazione:** si applica alle componenti specificate nella progettazione architetturale, si selezionano le funzionalità da integrare, si identificano le componenti che le svolgono e si ordina tali componenti per numero di dipendenze crescenti. Problemi rilevati in questo tipo di test sono dovuti a difetti di progettazione o insufficiente qualità dei test di qualità. La quantità è stabilita da quanti ne servono per accertare che tutti i dati scambiati attraverso un'interfaccia siano conformi alla specifica, e che tutti i flussi di controllo previsti in specifica siano stati realizzati e provati.

- **Test di sistema:** verificano il comportamento dinamico del sistema completo rispetto ai requisiti software, iniziano con il completamento dei test di integrazione.

- **Test di regressione:** possono essere molto onerosi, si accertano che le modifiche introdotte non comportino errori nel sistema, ripetizione selettiva di TS o TI.

- **Test di accettazione:** accertano il soddisfacimento dei requisiti utente.

Per una prova è interessante sapere il **Fattore di copertura** che è suddiviso in *Copertura funzionale* (rispetto alla percentuale di funzionalità esercitate come viste dall'esterno) e in *Copertura strutturale* (rispetto alla percentuale di logica interna del codice esercitata).