Definizione e Contesto

L'**analisi dinamica** è una delle due forme fondamentali di verifica del software, complementare all'analisi statica. Consiste nell'esecuzione di **test** (prove) su codice in esecuzione per verificare il comportamento dinamico del programma.

Caratteristiche Principali

- **Richiede esecuzione**: a differenza dell'analisi statica, necessita che il codice sia eseguibile
- **Rileva malfunzionamenti**: identifica la presenza di difetti attraverso l'osservazione del comportamento
- **Non esaustiva**: può essere applicata solo a un insieme finito di casi di prova
- **Tardiva nel ciclo di vita**: il codice eseguibile diventa disponibile in fasi avanzate dello sviluppo

Limiti Intrinseci

> **"Testing shows the presence, not the absence of bugs"** (Edsger W. Dijkstra)

L'analisi dinamica non può garantire l'assenza di difetti, ma può solo rilevarne la presenza. Questo limite deriva dall'impossibilità di testare tutte le possibili esecuzioni del programma, che spesso costituiscono un dominio infinito o comunque troppo vasto.

Terminologia e Gerarchia degli Errori

Catena Causale: Mistake → Fault → Error → Failure

La terminologia del testing si articola su una gerarchia causale:

1. Mistake (Errore Umano)

- **Definizione**: errore umano commesso durante lo sviluppo
- **Natura**: causa esterna al sistema
- **Origine**: programmatore, analista, progettista

2. Fault (Difetto/Guasto)

- **Definizione**: manifestazione concreta di un mistake nel sistema
- **Natura**: anomalia presente nel codice o nella documentazione
- **Tipologia**: può essere di natura algoritmica, logica, concettuale
- **Caratteristica**: può rimanere latente se non attivato

3. Error (Errore)

- **Definizione**: stato interno errato del sistema causato da un fault
- **Natura**: stato del sistema durante l'esecuzione
- **Attivazione**: si manifesta quando il fault viene attraversato durante l'esecuzione
- **Propagazione**: può rimanere nascosto se non produce effetti esterni osservabili

4. Failure (Malfunzionamento)

- **Definizione**: comportamento esterno del sistema difforme dalle attese
- **Natura**: effetto osservabile di un error
- **Rilevabilità**: è ciò che viene effettivamente rilevato durante i test
- **Impatto**: influenza direttamente l'utente o il committente

Nota: Il glossario IEEE presenta alcune variazioni rispetto a questa terminologia, ma la catena causale rimane concettualmente valida.

Componenti dell'Analisi Dinamica

Elementi di Base

Caso di Prova (Test Case)

Una tripla strutturata che specifica completamente un singolo test:

```
<Ingresso, Uscita Attesa, Ambiente>
```

Componenti:

- **Ingresso**: dati di input forniti al sistema
- **Uscita attesa**: risultato previsto (oracolo)
- **Ambiente**: configurazione hardware/software, stato iniziale, precondizioni

Batteria di Prove (Test Suite)

Un insieme (sequenza) di casi di prova correlati, organizzati per:

- Obiettivo comune
- Livello di testing (unità, integrazione, sistema)
- Area funzionale del sistema
- Priorità di esecuzione

Procedura di Prova

Il procedimento completo per:

- Eseguire la batteria di prove
- Registrare i risultati
- Analizzare gli esiti
- Valutare la conformità alle attese

Strumenti di Supporto

L'analisi dinamica richiede componenti di supporto per automatizzare e gestire l'esecuzione dei test:

1. Driver

- **Ruolo**: componente attiva fittizia
- **Funzione**: pilota l'esecuzione del test, simula il chiamante
- **Utilizzo**: necessario per testare unità che non hanno un entry point proprio
- **Natura**: "usa e getta", specifico per ogni test
- **Relazione**: aumenta il fan-in dell'unità testata

2. Stub

- **Ruolo**: componente passiva fittizia
- **Funzione**: simula parti del sistema non ancora implementate o dipendenze
- **Utilizzo**: sostituisce moduli chiamati dall'unità in test
- **Natura**: fornisce risposte predefinite senza implementare la logica reale
- **Relazione**: riduce il fan-out dell'unità testata
- **Dualità**: complementare al driver (driver simula chiamanti, stub simula chiamati)

3. Logger

- **Ruolo**: componente non intrusivo di registrazione
- **Funzione**: scrive automaticamente l'esito delle prove su file persistenti
- **Caratteristiche**:
  - Registrazione automatica e persistente
  - Non altera il comportamento del sistema testato
  - Permette analisi differita dei risultati
  - Fornisce tracciabilità completa dell'esecuzione

Principi Fondamentali del Testing

Principi di Bertrand Meyer

1. Testing come Ricerca Attiva di Fallimenti

> *"To test a program is to try to make it fail"*

- Il testing deve essere **"cinico"**: l'obiettivo è far fallire il software
- Un test è buono se rivela difetti
- Approccio opposto al debugging reattivo

2. Test vs Specifiche

> *"Tests are no substitutes for specifications"*

- I test non sostituiscono la progettazione
- Le specifiche guidano la creazione dei test
- I test verificano la conformità alle specifiche

3. Permanenza dei Test Falliti

> *"Any failed execution must yield a test case, permanently included in the project's test suite"*

- Ogni test fallito diventa permanente nella suite
- Previene la reintroduzione di difetti già risolti
- Costituisce documentazione viva del progetto

4. Oracoli come Contratti

> *"Oracles should be part of the program text, as contracts"*

- I comportamenti attesi devono essere espliciti
- L'oracolo decide l'esito dell'esecuzione
- Design by Contract integra verifica nel codice

5. Riproducibilità e Valutazione Oggettiva

> *"Any testing strategy should include a reproducible testing process and be evaluated objectively with explicit criteria"*

- Ogni test deve essere **ripetibile**
- Specifica completa: ingresso, uscita attesa, ambiente, stato iniziale
- Criteri di valutazione espliciti e misurabili

6. Efficacia Temporale della Strategia

> *"A testing strategy's most important quality is the number of faults it uncovers as a function of time"*

- La qualità si misura nel numero di difetti trovati nel tempo
- Strategie diverse hanno curve di efficacia diverse
- Il tempo è risorsa critica nel testing

Compromesso Economico del Testing

Legge dei Rendimenti Decrescenti (Diminishing Returns)

La strategia di testing deve bilanciare due esigenze contrapposte:

Quantità Minima di Test

- Sufficiente a fornire adeguata certezza sulla qualità
- Copre scenari critici e comuni
- Rispetta i vincoli di progetto

Quantità Massima di Test

- Limitata da sforzo, tempo e risorse disponibili
- Oltre una soglia, i test aggiuntivi trovano pochi o nessun difetto
- Il rendimento (difetti trovati / sforzo investito) decresce

Curva di Efficacia

Il comportamento tipico è:

1. **Fase iniziale**: alta produttività, molti difetti rilevati con poco sforzo
2. **Fase intermedia**: rendimento ancora positivo ma decrescente
3. **Fase finale**: costo dei test supera il valore dei difetti trovati

Ottimizzazione secondo Musa & Ackerman

Criterio di stop: bilanciare il costo dello sviluppo (incluso testing) con il costo d'uso derivante dai difetti residui.

Relazioni:

- **Costo d'uso**: cresce linearmente con l'intensità dei difetti residui
- **Costo di sviluppo**: cresce in modo non lineare con la qualità desiderata

Obiettivo: identificare la densità di difetti residui accettabile che minimizzi il costo totale.

Considerazioni Pratiche

> *"È interessante notare che il testing viene fatto sempre: se non lo fa il fornitore, lo farà l'utente."*

- Non testare significa trasferire il costo al cliente
- L'utente non è un verificatore: ogni difetto danneggia la reputazione
- Il testing interno è sempre più economico del testing in produzione

---

Criteri Guida per i Test

Oggetto della Prova

Il testing si applica a diversi livelli di granularità:

1. **Sistema completo**: per test di sistema e collaudo
2. **Parti del sistema**: componenti in relazione funzionale, d'uso, comportamentale o strutturale (test di integrazione)
3. **Singole unità**: moduli o aggregati di moduli (test di unità)

Obiettivi dei Test

Ogni test deve avere un obiettivo:

- **Chiaro**: univocamente definito e comprensibile
- **Utile**: significativo per la qualità del sistema
- **Decidibile**: esito verificabile tramite oracolo
- **Misurabile**: quantificabile in termini di copertura o conformità

Requisiti di Ripetibilità

Per essere ripetibile, un test deve specificare:

1. **Stato iniziale**: configurazione del sistema prima del test
2. **Ambiente di esecuzione**: hardware, software, configurazioni
3. **Dati di ingresso**: input precisi e completi
4. **Comportamenti attesi**: output e side-effects previsti
5. **Procedura**: passi per eseguire e analizzare il test

Pianificazione Anticipata

- Il testing deve essere pianificato **il prima possibile**
- Il primo momento utile è la **progettazione del sistema**
- Specifica nel modello a V:
    - Test di sistema e validazione → durante analisi dei requisiti

- Test di integrazione → durante progettazione architetturale
- Test di unità → durante progettazione di dettaglio

Debugging vs Verifica

Distinzione fondamentale:

- **Debugging**: reattivo, nasce da un errore inaspettato
- **Verifica**: proattiva, pianificata a monte dello sviluppo

---

Tassonomia dei Test

Test di Unità (TU)

Caratteristiche

- **Granularità**: più piccola quantità di software verificabile autonomamente
- **Definizione**: avviene durante la progettazione di dettaglio
- **Parallelismo**: eseguiti con alto grado di parallelismo (riflette parallelismo dei programmatori)
- **Rilevamento**: circa **2/3 dei difetti** complessivi trovati dall'analisi dinamica
- **Responsabilità**: programmatore, verificatore indipendente, o automazione (preferibile)

Unità Software

Un'**unità** è composta da uno o più **moduli**, dove il modulo è il componente elementare della progettazione di dettaglio. La definizione di unità dipende dal linguaggio e dall'architettura:

- Funzione/procedura
- Classe
- Package/modulo
- Componente con interfaccia ben definita

Concetto di Copertura (Coverage)

La **copertura** misura la percentuale di codice attraversato durante l'esecuzione dei test.

Criteri di copertura principali:

1. **Function Coverage**
   - Quante funzioni/sottoprocedure sono state eseguite
   - Minimo accettabile: tutte le funzioni pubbliche almeno una volta
2. **Statement Coverage (C0)**
   - Quante istruzioni (statement) sono state eseguite

- Obiettivo: 100% delle linee eseguibili
- Limite: non verifica le decisioni logiche

3. **Branch Coverage (C1)**
   - Quanti rami decisionali (then/else) sono stati attraversati
   - Criterio più forte di statement coverage
   - Obiettivo: 100% dei branch
   - **Fondamentale**: sempre richiesto nei test di unità professionali
4. **Condition Coverage**
   - Quante condizioni atomiche hanno assunto entrambi i valori (true/false)
   - Più preciso del branch coverage in presenza di espressioni complesse
5. **MC/DC (Modified Condition/Decision Coverage)**
   - Ogni condizione deve influenzare indipendentemente il risultato della decisione
   - Criterio più forte, richiesto in sistemi critici (avionica, medicale)
   - Standard in DO-178C per software avionic

Complessità Ciclomatica

Il numero di percorsi linearmente indipendenti è dato dalla **complessità ciclomatica** (CC):

```
CC = e − n + 2p
```

Dove:

- **e**: numero di archi nel grafo (flussi tra comandi)
- **n**: numero di nodi (espressioni o comandi)
- **p**: numero di componenti connesse (normalmente 2: entry e exit)

L'esecuzione sequenziale ha p=2 (un predecessore e un successore per ogni arco).

Interpretazione:

- CC indica il numero minimo di test per branch coverage al 100%
- CC inizialmente 1, incrementata da branch, salti, iterazioni

Categorie di Test di Unità

Test Funzionale (Black-Box)

Caratteristiche:

- Fa riferimento **solo alla specifica** dell'unità
- Osserva il comportamento dall'esterno, ignorando l'implementazione
- Utilizza dati di ingresso che provocano specifici comportamenti funzionali

- **Classi di equivalenza**: dati che producono lo stesso comportamento costituiscono un caso di prova

Classi di equivalenza tipiche:

1. Valori non ammessi (fuori dominio)
2. Valori barriera (boundary values)
3. Valori ammessi entro barriere

Contributo:

- Accumula **requirements coverage**: misura di quanti requisiti funzionali sono soddisfatti
- Non valuta la logica interna dell'unità

Limiti:

- Incapace di accertare correttezza e completezza della logica interna
- Non rileva codice non necessario o ridondante
- Necessita integrazione con test strutturali

Test Strutturale (White-Box)

Caratteristiche:

- Verifica la **logica interna** del codice dell'unità
- Persegue massima **structural coverage** (complementare a requirements coverage)
- Ogni caso di prova attiva uno specifico cammino d'esecuzione
- Creazione delle condizioni logiche che causano la scelta di quel cammino

Costituzione del caso di prova:

- Insieme di dati di ingresso
- Configurazione di ambiente
- Che producono uno specifico cammino d'esecuzione

Contributo:

- Assicura che tutto il codice scritto sia stato esercitato
- Rileva codice morto (unreachable)
- Identifica percorsi logici non previsti

Supporto:

- I debugger facilitano l'esecuzione ma non esonerano dalla progettazione dei casi di prova
- Strumenti di code coverage automatizzano la misurazione

Ordine di Esecuzione

> **Regola fondamentale**: eseguire **prima** i test funzionali, **poi** quelli strutturali.

Motivazione: evitare di analizzare una struttura che non svolge il compito corretto.

Sinergia: i test funzionali vanno **sempre integrati** con test strutturali per una verifica completa.

Test di Integrazione (TI)

Contesto e Obiettivi

I test di integrazione sono parte del processo più ampio di **integrazione delle componenti** del sistema.

Scopo:

- Verificare il sistema in modo **incrementale**
- Rilevare difetti di **progettazione architetturale**
- Accertare la corretta comunicazione tra componenti

Oggetto

Si applica alle **componenti** specificate nella progettazione architetturale:

- Moduli
- Sottosistemi
- Servizi
- Componenti con interfacce definite

L'integrazione totale delle componenti costituisce il **sistema completo**.

Quanti Test di Integrazione?

Criterio quantitativo: tanti quante sono le **interfacce** nell'architettura del sistema.

Obiettivi verificati:

1. Tutti i dati scambiati attraverso ciascuna interfaccia aderiscono alla propria specifica
2. Tutti i flussi di controllo previsti in specifica sono stati effettivamente realizzati
3. Le assunzioni fatte da ciascun componente sulle sue dipendenze sono valide

Problemi Rilevati

I test di integrazione identificano:

- **Errori residui** nella realizzazione delle componenti
- **Modifiche delle interfacce** non propagate correttamente
- **Cambiamenti nei requisiti** non riflessi in tutte le componenti
- **Riuso di componenti** dal comportamento oscuro o inadatto
- **Integrazione con applicazioni esterne** non ben conosciute

Principi di Integrazione

Incrementalità

Regola: assemblare moduli in modo **incrementale**

Benefici:

- I difetti rilevati sono **più probabilmente attribuibili** al modulo ultimo aggiunto
- Facilita la localizzazione degli errori
- Riduce la complessità del debugging

Reversibilità

Principio: ogni passo di integrazione deve essere **reversibile**

Implementazione:

- Mantenere stati sicuri verificati
- Possibilità di retrocedere a configurazione precedente stabile
- Version control e configurazione baseline

Strategie di Integrazione

I sistemi software moderni sono **gerarchici** (ad albero di dipendenze). Le due strategie principali riflettono questa struttura.

Bottom-Up (Dal Basso)

Approccio:

- Si sviluppano e integrano **prima** le componenti con:
  - **Minore dipendenza funzionale** (basso fan-out)
  - **Maggiore utilità interna** (alto fan-in)
- Componenti molto chiamate/attivate ma che chiamano/attivano poco
- Componenti più interne al sistema, meno visibili all'utente

Percorso: si "risale" l'albero delle dipendenze dalla persistenza verso l'interfaccia utente.

Vantaggi:

- **Economizza stub**: le componenti di basso livello sono reali
- Le fondamenta sono solide prima di costruire sopra
- Facilita il riuso di componenti di base

Svantaggi:

- **Ritarda funzionalità visibili**: l'utente vede risultati tardi
- Difficile mostrare progress al committente
- Richiede molti driver per testare componenti di basso livello

Quando usarla:

- L'utente non ha ancora un punto di partenza operativo
- Focus su solidità delle fondamenta
- Componenti di basso livello critiche per il sistema

Top-Down (Dall'Alto)

Approccio:

- Si sviluppano e integrano **prima** le componenti con:
  - **Maggiori dipendenze d'uso** (alto fan-out)
  - **Maggiore valore aggiunto esterno** (visibilità utente)
- Componenti che chiamano/attivano più di quanto siano chiamate
- Funzionalità di alto livello, direttamente percepibili dall'utente

Percorso: si "scende" l'albero delle dipendenze dall'interfaccia utente verso la persistenza.

Vantaggi:

- **Funzionalità visibili subito**: si può mostrare una bozza al committente
- Integra prima le funzionalità di più alto livello
- Facilita negoziazione e feedback del cliente
- Richiede pochi driver

Svantaggi:

- **Richiede molti stub**: componenti di basso livello simulate
- Gli stub possono mascherare problemi nelle componenti reali
- Rischio di scoprire tardi problemi architetturali fondamentali

Quando usarla:

- Utile negoziare frequentemente con l'utente
- Necessità di dimostrare progress visibile
- Approccio iterativo/agile con feedback continuo

Approccio Sandwich (Ibrido)

Combinazione:

- Integrazione simultanea bottom-up e top-down
- Incontro nel mezzo dell'architettura
- Più frequente nella pratica

Vantaggi:

- Minimizza sia driver che stub
- Ottimizza tempi di integrazione
- Bilancia visibilità utente e solidità fondamenta

Logica di Integrazione Funzionale

Processo sistematico per l'integrazione:

1. **Selezione**: identificare le funzionalità da integrare
2. **Identificazione**: individuare le componenti che svolgono quelle funzionalità
3. **Ordinamento**: ordinare le componenti per numero di dipendenze (crescente per bottom-up, decrescente per top-down)
4. **Integrazione**: eseguire l'integrazione nell'ordine stabilito
5. **Verifica**: testare l'incremento prima di procedere

Test di Sistema (TS)

Caratteristiche Fondamentali

Definizione: verifica il comportamento dinamico del sistema completo rispetto ai requisiti software.

Timing:

- Inizia al **completamento del test di integrazione**
- Precede il collaudo (test di accettazione)

Natura: inerentemente **funzionale** (black-box)

- Non richiede conoscenza della logica interna del sistema
- Focus sulla conformità ai requisiti, non sull'implementazione
- Analogamente ai requisiti funzionali: fissa aspettativa, non realizzazione

Obiettivi

1. **Accertare la copertura dei requisiti** (requirements coverage)
2. Verificare requisiti funzionali e non funzionali
3. Validare comportamento end-to-end del sistema
4. Confermare che il sistema è pronto per il collaudo

Raccomandazione di Meyer:

> I test di sistema devono includere **tutti i casi di prova precedentemente falliti**

Questo assicura non-regressione e tracciabilità dei difetti storici.

Requirements Coverage

Misura cumulativa:

- Si basa sulla requirements coverage accumulata dai test di unità funzionali
- Estende la verifica a livello di sistema integrato
- Ogni requisito deve essere coperto da almeno un test di sistema

Responsabilità

- Attività **interna all'organizzazione** (fornitore)
- Condotta da team di QA dedicato
- Separazione dai programmatori (indipendenza)

Test di Regressione (TR)

Definizione

Ripetizione **selettiva** di test di unità, integrazione e sistema per accertare che modifiche (correzioni o estensioni) non abbiano introdotto errori in parti del sistema precedentemente funzionanti.

Obiettivi

- Verificare che correzioni su una parte P di S non causino errori in P
- Verificare che correzioni non causino errori in parti del sistema in relazione con S
- Prevenire la "regressione" della qualità dopo modifiche

Quando Eseguire

Dopo ogni:

- Correzione di difetti
- Estensione funzionale

- Refactoring significativo
- Modifica di interfacce

Composizione

Include:

- Test già specificati e già eseguiti (test suite esistente)
- Nuovi test specifici per la modifica effettuata

Selettività:

- Non sempre necessario rieseguire tutti i test
- Analisi di impatto identifica test rilevanti per la modifica
- Bilanciamento tra copertura e costo

Gestione della Complessità

La regressione è **complicata** e richiede:

- Analisi accurata delle dipendenze
- Tracciabilità tra codice e test
- Automazione estensiva della suite
- Studio ad hoc della strategia di selezione

Fattori di riduzione:

- **Basso accoppiamento**: limita propagazione di modifiche
- **Incapsulamento**: isola impatti delle modifiche
- **Architettura modulare**: facilita analisi di impatto

Coinvolgimento Processi

Il test di regressione coinvolge:

1. **Problem Resolution**: valuta necessità di modifiche (correttive o adattative) e le approva
2. **Change Management**: gestisce la buona realizzazione delle modifiche approvate

Entrambi i processi assicurano che le modifiche siano controllate e verificate.

Test di Accettazione / Collaudo (TA)

Definizione

Verifica finale che accerta il **soddisfacimento dei requisiti utente** alla presenza del committente.

Caratteristiche

Natura:

- Attività **esterna** supervisionata dal committente
- Duale del test di sistema (interno vs esterno)
- Casi di prova definiti nel **contratto**

Valore contrattuale:

- Attività formale con implicazioni legali
- Segue il **rilascio del sistema**
- Conclusione della commessa (salvo manutenzione e garanzie)

Varianti

1. **UAT (User Acceptance Testing)**: eseguito dagli utenti finali effettivi
2. **Alpha Testing**: eseguito internamente dall'organizzazione in ambiente controllato
3. **Beta Testing**: eseguito da utenti esterni in ambiente reale di produzione

Esito

- **Successo**: accettazione formale del sistema, chiusura commessa
- **Fallimento**: necessità di correzioni, nuovo ciclo di verifica, penali contrattuali

---

Sintesi: Modello a V

Il **modello a V** illustra la corrispondenza tra fasi di sviluppo (ramo discendente) e fasi di verifica (ramo ascendente):

Ramo Discendente (Sviluppo)

1. **Capitolato → Analisi dei Requisiti**
   - Definisce **cosa** il sistema deve fare
   - Specifica requisiti funzionali e non funzionali
   - **Corrisponde a**: Test di Sistema e Collaudo
2. **Progettazione Architetturale (Logica)**
   - Definisce **come** il sistema è strutturato in componenti
   - Specifica interfacce e dipendenze
   - **Corrisponde a**: Test di Integrazione
3. **Progettazione di Dettaglio**
   - Definisce implementazione delle singole unità
   - Specifica algoritmi e strutture dati

    - **Corrisponde a**: Test di Unità
4. **Codifica**
    - Implementazione effettiva
    - Centro del modello a V

Ramo Ascendente (Verifica)

1. **Test di Unità** (TU)
    - Verificano progettazione di dettaglio
    - Parallelismo elevato
2. **Test di Integrazione** (TI)
    - Verificano progettazione architetturale
    - Integrazione incrementale (Build)
3. **Test di Sistema** (TS)
    - Verificano analisi dei requisiti
    - Requirements coverage
4. **Collaudo** (TA)
    - Valida conformità al capitolato
    - Presenza committente

Tracciamento

Il **tracciamento** (tracing) collega ogni fase di sviluppo alla corrispondente fase di verifica:

- Assicura completezza: tutti i requisiti sono verificati
- Assicura coerenza: ogni test verifica elementi specificati
- Assicura economicità: nessuna funzionalità superflua, nessun test ingiustificato

---

Misure di Copertura

Definizione e Importanza

Le **misure di copertura** quantificano quanto le prove esercitano il prodotto software:

- **Copertura funzionale**: rispetto ai requisiti del sistema (requirements coverage)
- **Copertura strutturale**: rispetto alla logica interna del codice (structural coverage)

Scopo: quantificare la **bontà della campagna di test**.

Limiti delle Misure

Impossibilità del 100%

Raggiungere il 100% di copertura può essere **complesso o impossibile**:

Ragioni:

- Vincoli di tempo e costo
- Complessità della codifica (codice morto, percorsi non raggiungibili)
- Limiti degli strumenti di misurazione
- Condizioni di esecuzione non riproducibili

Copertura ≠ Assenza di Difetti

> **Principio fondamentale**: raggiungere il 100% di copertura **non garantisce** l'assenza di difetti

Perché:

- Un percorso coperto può comunque contenere logica errata
- I dati di test potrebbero non rivelare difetti specifici
- Difetti di integrazione o di sistema possono esistere nonostante copertura unitaria completa

Relazione Rischio-Copertura

Principio: più alta è la copertura, più basso è il rischio di difetti non rilevati.

La copertura è una metrica di **confidenza**, non di **correttezza assoluta**.

Specifica degli Obiettivi

Gli **obiettivi di copertura** sono specificati nel **Piano di Qualifica** (PdQ):

- Percentuale minima richiesta per statement coverage
- Percentuale minima richiesta per branch coverage
- Criteri specifici per componenti critiche (es. MC/DC per safety-critical)
- Giustificazione per esclusioni (codice non eseguibile, casi impossibili)

Criteri Operativi

Ripetibilità e Riproducibilità

Requisito fondamentale: i test devono essere **ripetibili**

Per assicurare ripetibilità, i test specificano:

- **Ambiente d'esecuzione**: hardware, software, configurazione di rete
- **Stato iniziale**: configurazione del sistema, dati preesistenti

- **Ingressi richiesti**: input precisi e completi
- **Uscite ed effetti attesi**: output e side-effects previsti
- **Procedure**: esecuzione, analisi dei risultati

Automazione: da "Codifica" in poi si vuole **automazione completa**:

1. Specifica (manuale)
2. Codifica (automatizzata)
3. Compilazione (automatizzata)
4. Esecuzione (automatizzata)
5. Analisi (automatizzata, logger)

Costo e Risorse

Realtà: le prove sono **costose**

Richiedono:

- Tempo: specifica, implementazione, esecuzione, analisi
- Persone: verificatori, analisti, automazione test
- Infrastruttura: ambienti di test, strumenti, licenze

Governo: vanno gestite con **efficienza ed efficacia**

Richiedono cicli completi di:

- Analisi (pianificazione)
- Progettazione (casi di prova)
- Codifica (implementazione test)
- Esecuzione (campagne di test)

Efficienza vs Efficacia

Compromesso necessario:

- **Efficienza**: massimizzare difetti trovati / risorse impiegate
- **Efficacia**: massimizzare copertura e confidenza nella qualità

Bilanciamento:

- Prioritizzare test ad alto valore (funzionalità critiche, percorsi comuni)
- Automazione per ridurre costi di esecuzione ripetuta
- Test-driven development per integrare testing nello sviluppo

---

Riferimenti

Letture Fondamentali

1. **J.D. Musa, A.F. Ackerman**, *Quantifying software validation: when to stop testing?*, IEEE Software, maggio 1989
   - Modello economico per decidere quando terminare il testing
   - Analisi costi-benefici del testing
2. **B. Meyer**, *Seven Principles of Software Testing*, IEEE Computer, agosto 2008
   - Principi fondamentali della disciplina
   - Filosofia del testing efficace
3. **J.C. Westland**, *The cost of errors in software development: evidence from industry*, Journal of Systems and Software 62(1)
   - Analisi empirica del costo dei difetti
   - Evidenze industriali sulla qualità
4. **G.J. Myers**, *The Art of Software Testing*, Wiley
   - Testo classico sul testing software
   - Tecniche e best practices

Standard

- **British Computer Society SIGIST**, *Standard for Software Component Testing*, 1997
- **M.E. Fagan**, *Advances in Software Inspection*, IEEE Transaction on Software Engineering, luglio 1986
- **G.A. Cignoni, P. De Risi**, *Il test e la qualità del software*, Il Sole 24 Ore, 1998

Conclusioni

L'analisi dinamica è disciplina complessa che richiede:

1. **Pianificazione anticipata**: definire test durante progettazione
2. **Approccio sistematico**: seguire il modello a V e principi consolidati
3. **Bilanciamento economico**: ottimizzare costi vs benefici secondo legge dei rendimenti decrescenti
4. **Automazione estensiva**: ridurre costi di esecuzione ripetuta
5. **Complementarità**: integrare con analisi statica per copertura completa
6. **Realismo**: accettare limiti intrinseci (non-esaustività) e gestire rischio residuo

> **Principio finale**: il testing non garantisce assenza di difetti, ma fornisce **misura di confidenza** nella qualità del software. La disciplina ingegneristica consiste nel massimizzare questa confidenza entro vincoli di progetto.