

*Corso di Laurea in Ingegneria Informatica*

# Corso di Ingegneria del Software

---

## Verifica e Validazione

# Sommario

- Verifica e Validazione del software
- Tipologie di testing.
- Il processo di qualità. Principi
- Obiettivi di qualità. Pianificazione e monitoraggio

## Riferimenti

M.Pezzè, M. Young; *Software Testing and Analysis. Process, principles, and techniques.*

**Cap.** 1, 2, 3, 4

C.Ghezzi, M. Jazayeri, D. Mandrioli; *Ingegneria del Software. Fondamenti e principi*, II edizione. Pearson.

**Cap 6, § § 6.1, 6.2, 6.3, 6.4**

# La verifica nei processi di ingegnerizzazione del software

- ❖ Ogni disciplina ingegneristica associa attività di “costruzione” con attività di verifica degli artefatti intermedi e del prodotto finale
- ❖ L’ingegneria del software non fa eccezione: la realizzazione di prodotti software di qualità richiede
  - **Costruzione** e
  - Attività di **verifica**
- ❖ Nell’ingegneria del software, come in ogni campo dell’ingegneria, l’ausilio di *tool* sofisticati
  - Amplifica, migliora le capacità, ma ...
  - ... non rimuove del tutto gli errori umani

# Progettazione e Verifica

- La progettazione e la verifica possono assumere diverse forme, a seconda, ad es., che si riferiscano a:
  - Attività di costruzione altamente ripetitive di prodotti non critici per un mercato di massa
  - Costruzione di prodotti altamente personalizzati, o di prodotti critici per un mercato ristretto o di nicchia
- Una appropriata attività di verifica dipende da:
  - La **disciplina ingegneristica** che si sta considerando
  - Il **processo di costruzione**
  - Il **prodotto finale**
  - I **requisiti di qualità**

# Peculiarità del software

- ♦ Il software ha delle peculiarità che rendono l' attività di verifica e validazione particolarmente difficile:
  - Molti **requisiti**, spesso contrastanti, **di qualità**
  - Una **struttura evolutiva** (e che si “deteriora”)
  - Una “**non-linearità**” intrinseca
    - ◆ Ad es., se *un ascensore può portare un carico di 1000Kg, può anche portare un qualsiasi carico minore; ma se una procedura ordina correttamente un set di 256 elementi, essa può fallire su 255, o 53, o 12, come pure su 257 o 1023*
  - Una **distribuzione** dei guasti **irregolare**

# Varietà di approcci

- ◆ Non ci sono ricette fisse!
- ◆ I progettisti dei test devono
  - Scegliere e schedulare la giusta combinazione di tecniche
    - ◆ Per raggiungere il livelli di qualità richiesto
    - ◆ Entro i vincoli di costo definiti
  - Progettare una soluzione specifica, adatta
    - ◆ Al problema
    - ◆ Ai requisiti
    - ◆ All' ambiente di sviluppo

# Verifica e Validazione (V&V)

- ♦ **Validazione**: il sistema software soddisfa i bisogni reali dell' utente?
  - *are we building the right software?*
  
- ♦ **Verifica**: il software soddisfa la specifica dei requisiti?
  - *are we building the software right?*

# Validazione

## Requisiti informali d' utente

Si parla in tal caso di validazione del software prodotto a fronte delle funzionalità richieste

La validazione esamina se il sistema software risolve il problema applicativo per cui è stato prodotto

**Data la informalità dei requisiti d' utente, la sola validazione non garantisce la correttezza del software**

# Verifica

Specifiche formali prodotte dall'analista

Si parla in tal caso di **verifica**, intendendo l'insieme delle attività tese a stabilire se il software è corretto rispetto alle specifiche dell'analista

È possibile definire tecniche precise per la verifica del software

# Definizioni (1)

- Il **Testing** (collaudo) è un processo di esecuzione del software allo scopo di scoprirne i malfunzionamenti
- Il **Debugging** è il processo di scoperta delle anomalie a partire da malfunzionamenti noti
- Esso isola e rimuove le cause del malfunzionamento (*difetti* software)
- **Test Case** (caso di test): un insieme di input, condizioni di esecuzione, e un criterio di *pass/fail*
- **Test Case Specification**: un requisito che deve essere soddisfatto da uno più casi di test
- **Test suite**: un insieme di casi di test. Può essere fatta da diverse suite per singoli moduli, sottosistemi o caratteristiche

# Definizioni (2)

- **Test obligation**: una specifica dei casi di test *parziale*, che richiede qualche proprietà ritenuta importante per il test completo
- **Test o test execution**: l' attività di esecuzione dei casi di test e valutazione dei loro risultati
- **Oracolo**: Descrizione del comportamento atteso, che si applica a una esecuzione del programma per valutare secondo il criterio *pass/fail* se il comportamento osservato coincide con quello atteso
- **Criterio di adeguatezza**: un predicato su una coppia <programma, test suite>. Di solito, è espresso nella forma di una regola per derivare un insieme di *test obligation*. Il criterio è soddisfatto se ogni *test obligation* è soddisfatta da almeno un caso di test nella test suite.

# Il *testing* – 1/5

- ◆ Un buon caso di prova (**test-case**) è quello che ha una elevata probabilità di esporre malfunzionamento non noti
- ◆ Un **test-case** ha successo se scopre un malfunzionamento non ancora scoperto
- ◆ La progettazione dei **test-case** mira a definire casi di prova che scoprano sistematicamente differenti tipi di malfunzionamento con il minimo sforzo

# Il *testing* – 2/5

## Indirettamente il testing

- dimostra che il software rispetta le specifiche ed i requisiti
- fornisce un' indicazione sulla sua affidabilità e qualità globale

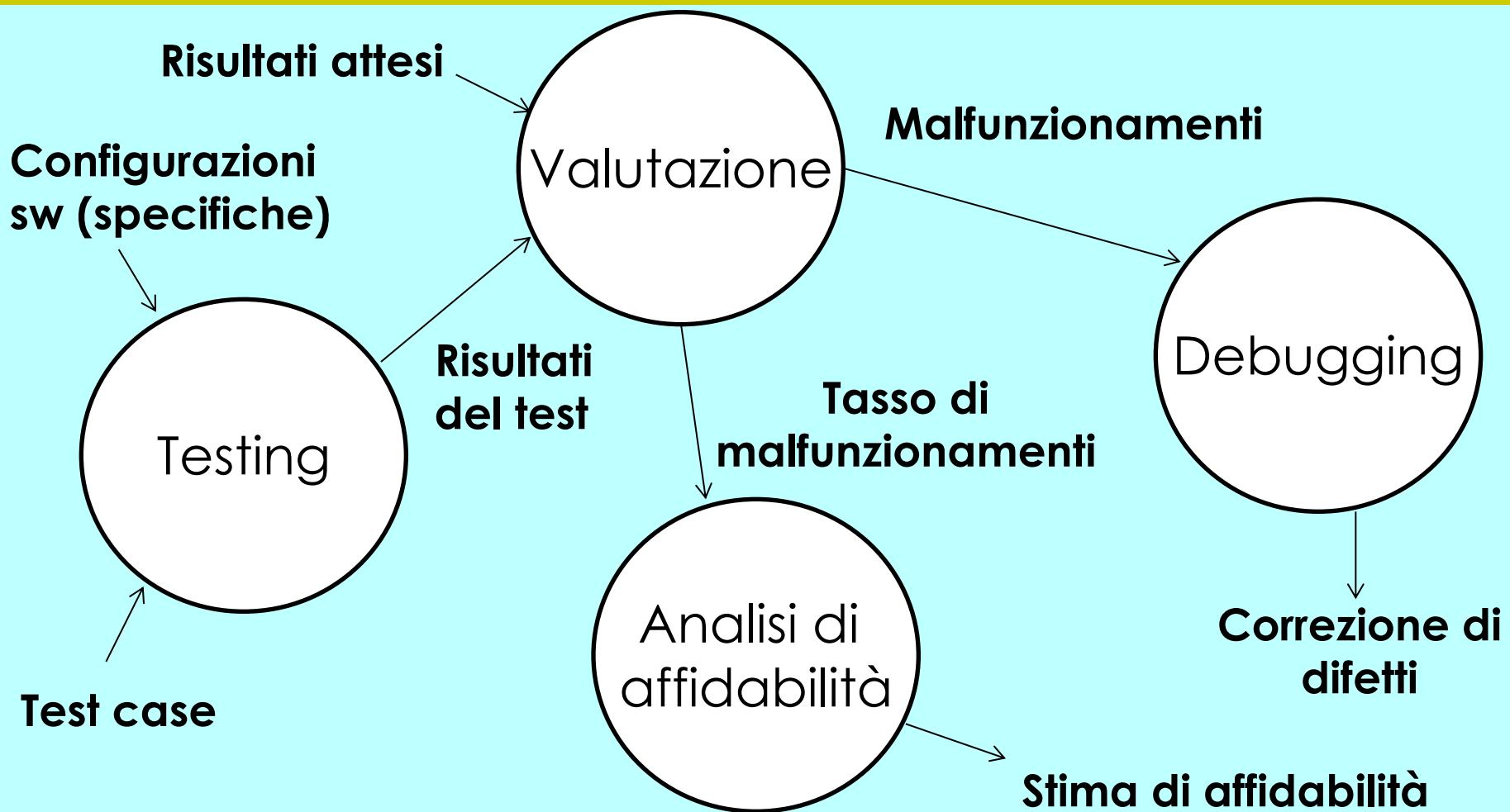
ma

- non può dimostrare l' assenza di difetti, può solo dimostrare la loro presenza (**Tesi di Dijkstra**)

# Il *testing* – 3/5

- ♦ La rilevazione di malfunzionamenti facilmente correggibili e/o il buon funzionamento del software possono indicare:
  - *affidabilità e qualità accettabili, o*
  - *inadeguatezza dei casi di prova (test-case)*
- ♦ Tali malfunzionamenti saranno poi riscontrati dagli utenti e ciò comporta un aggravio dei costi rispetto a quelli, già alti, della loro rilevazione in fase di sviluppo

# Il testing – 4/5



# Il *testing* – 5/5

- ♦ Ogni bolla indica un' attività molto complessa
- ♦ È difficile stimare la durata del testing
  - un malfunzionamento che mostra una pur lieve differenza con i risultati attesi può richiedere ore oppure settimane per la diagnosi e correzione del difetto che lo causa
- ♦ La rilevazione con una certa regolarità di malfunzionamenti gravi, che richiedono pesanti modifiche alla progettazione, mette in discussione l' affidabilità e la qualità
  - in tal caso è consigliabile effettuare ulteriori test

# Tecniche di verifica – 1/2

## ◆ Testing

- Volto a rilevare malfunzionamenti

## ◆ Analisi statica

- prende in esame le istruzioni del programma (che sono un insieme finito), senza eseguirlo
- è impossibile rilevare malfunzionamenti che dipendono dal valore assunto dinamicamente dalle variabili
- è più facile rilevare anomalie legate alla presenza di variabili non dichiarate in linguaggi fortemente tipizzati quali Pascal, Ada, C++, etc.

# Tecniche di verifica - 2/2

## ♦ Analisi dinamica

- prende in esame l' insieme delle esecuzioni di un programma (quasi mai limitato)
- vi è il problema di ridurre il numero di casi da esaminare

**Le varie tecniche sono spesso usate  
in maniera complementare**

# Analisi

- ◆ Le tecniche di analisi includono
  - **Tecniche di ispezione manuale**
  - **Analisi automatica**
- ◆ L' analisi mira a verificare il soddisfacimento di una proprietà
- ◆ Può essere applicata ad ogni fase del processo di sviluppo
- ◆ È particolarmente idonea nelle prime fasi, nella specifica e nella progettazione

# Ispezione

- Può essere applicata ad ogni documento
  - Requisiti
  - Documenti di progetto
  - Piani di test e casi di test
  - Codice sorgente
- Richiede una notevole quantità di tempo
  - Re-ispezionare un componente che è cambiato può essere molto costoso
- Usata principalmente
  - Quando le altre tecniche sono inapplicabili
  - Quando le altre tecniche non forniscono una sufficiente copertura

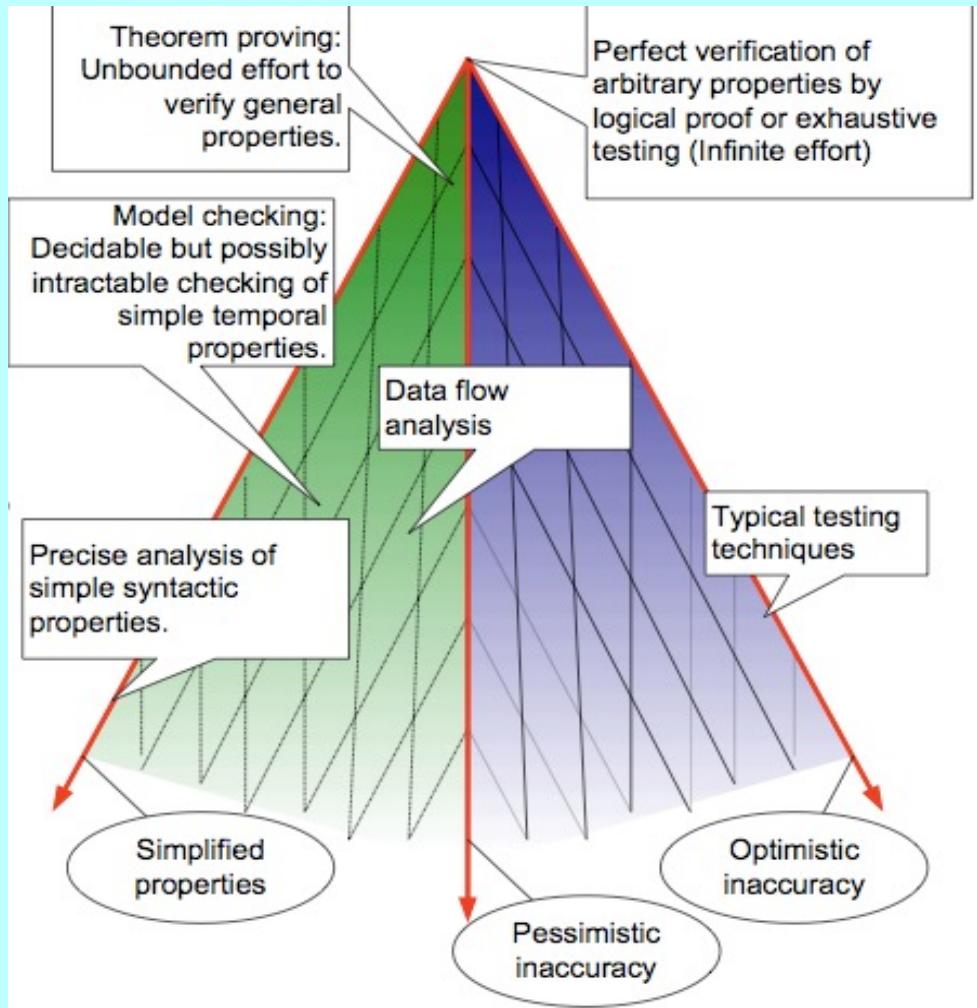
# Analisi statica

- ♦ Di applicabilità più limitata
  - Può essere applicata a rappresentazioni formali dei requisiti
  - Non a documenti espressi in linguaggio naturale
- ♦ Quando può essere applicata, è particolarmente efficiente, perchè consente di sostituire cicli macchina all' intervento umano per attività molto ripetitive e *error-prone*

# Testing

- Eseguito ad uno stadio avanzato del processo di sviluppo
- Inizia il prima possibile con la pianificazione dei test
- La generazione dei casi test nelle prime fasi ha diversi vantaggi:
  - Test generati indipendentemente dal codice, quando le specifiche sono ancora ben impresse nella mente dell' analista
  - La generazione dei casi di test può evidenziare inconsistenza e incompletezze delle specifiche corrispondenti
  - I test possono essere usati come compendio delle specifiche dai programmatori

# Punti di vista sulle tecniche



- **Inaccuratezza Ottimistica:** si accettano programmi che non posseggono la proprietà analizzata (i.e., possiamo non rilevare tutte le violazioni)
  - Testing
- **Inaccuratezza pessimistica:** non è garantita l' accettazione del programma anche se possiede la proprietà analizzata
  - Tecniche di analisi automatica
- **Proprietà semplificate:** riduce il grado di libertà per semplificare la proprietà da controllare

# Testing black-box e white-box

Vi sono due approcci generali al testing:

## ◆ **BLACK-BOX TESTING**

- Conoscendo le specifiche funzionali, effettuare test per dimostrare che ciascuna funzione è completamente operativa, ignorando i dettagli interni

## ◆ **WHITE-BOX TESTING**

- Conoscendo il funzionamento interno dei vari componenti, effettuare test per verificare che il meccanismo interno funzioni bene, cioè che le operazioni interne siano svolte correttamente

# Testing white-box

- ◆ Sono esaminati i dettagli del codice
- ◆ Viene svolto un test dei cammini logici interni
  - Test-case che esercitano insiemi specifici di condizioni e/o cicli
- ◆ Impossibilità in generale di eseguire un testing esaustivo di tutti i cammini, che può essere enormemente elevato anche per piccoli programmi

# Esempio

```
Repeat
  B0
  if R1 then
    if R2 then
      if R3 then
        B1
      else
        B2
      endif
    if R4 then
      B3
    else
      B4
    endif
  else B5
  endif
endif
until R6
```

- ◆ Se il ciclo viene eseguito max 20 volte si possono avere oltre 100.000 miliardi di cammini possibili
- ◆ 3.170 anni nell' ipotesi che ciascun test case sia elaborato in 1ms

# Testing black-box

- I test sono condotti sulle interfacce software per verificare che
  - I dati di input siano accettati in modo appropriato
  - I dati di output siano corretti
  - Sia mantenuta integrità delle informazioni esterne (es.: archivi di dati)
- Sono esaminati alcuni aspetti del modello fondamentale del sistema, senza preoccuparsi della struttura logica interna del software

# Testing

- ◆ **Test d' unità**
- ◆ **Test d' integrazione**
- ◆ **Test di sistema**
- ◆ **Test di validazione**

Quattro passi eseguiti in sequenza

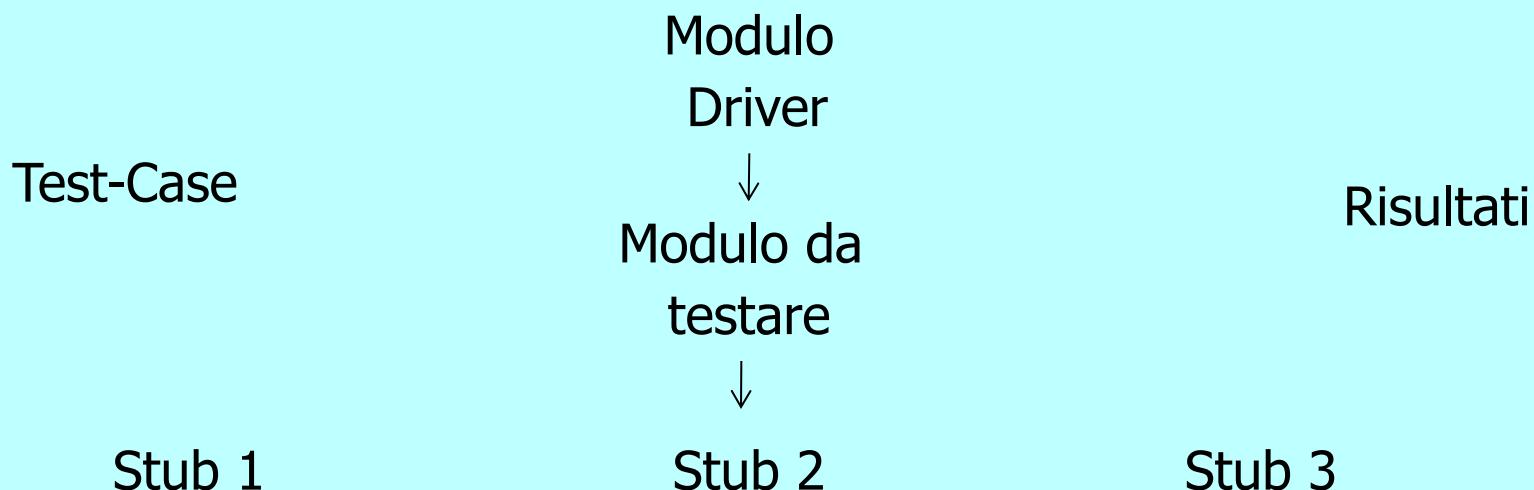


# Test di unità

Verifica della più piccola unità di progettazione software: **il modulo**

Ciascun modulo viene verificato sulla base della documentazione relativa alla progettazione dettagliata.

Spesso è effettuato subito dopo la codifica e ha necessità di utilizzare moduli pilota e filtri



# Procedure per il testing di unità

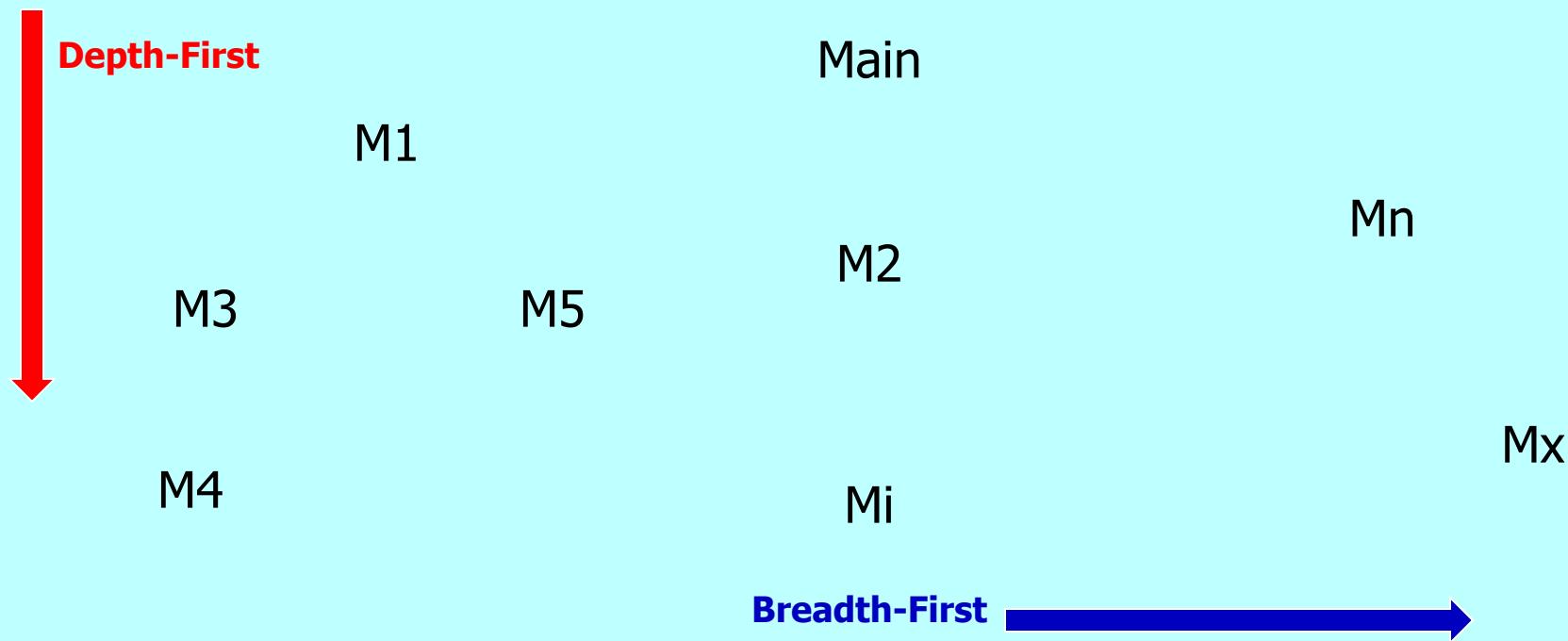
- ◆ Il **driver** simula i moduli chiamanti; esso accetta i dati dei test-case, li passa al modulo e stampa i risultati
- ◆ I moduli fintizi (**stub**) simulano i moduli gerarchicamente inferiori del modulo da testare, eseguono eventualmente qualche manipolazione di dati, stampano o restituiscono i dati nel modulo da testare
- ◆ Driver e stub sono onerosi in quanto devono essere realizzati ma non fanno parte del prodotto finale
- ◆ Talvolta alcuni moduli non possono essere testati con il solo uso di driver e stub; in tal caso il testing è posposto sino al testing di integrazione
- ◆ Testing di unità più facile con moduli ad alta coesione: minor numero di test-case, errori rilevabili più facilmente

# Test di integrazione

- ◆ Test per rilevare errori relativi all' interazione tra moduli
- ◆ Prendere i moduli testati singolarmente e costruire in maniera incrementale la struttura del programma
- ◆ Aggiungere uno o pochi moduli alla volta
- ◆ Perché è necessario, visto che i singoli moduli sono stati testati e corretti?
  - Scovare errori dovuti alla comunicazione (reale) tra i moduli, ovvero a problemi nelle interfacce
  - Effetti collaterali negativi di un modulo su un altro
  - Imprecisione, accettabile localmente, che si amplifica in modo intollerabile
  - Problemi per dati globali

# Integrazione top-down – 1/4

L'integrazione avviene seguendo la gerarchia di controllo iniziando dal MAIN ed aggregando man mano i moduli subordinati, con modalità *depth-first* o *breadth-first*



# Integrazione **top-down** – 2/4

Nell'approccio **depth-first** il cammino principale lungo cui effettuare l'integrazione è arbitrario.

Es: Main – M1 - M3 ... per i primi poi M4 oppure M5 se è necessario per il corretto funzionamento M1.

Si proseguirebbe con il cammino centrale con radice in M2 e così via.

Nell'approccio **breadth-first** i moduli vengono aggregati muovendosi orizzontalmente in ciascun livello gerarchico:

Prima M1-M2 .... Mn

Poi M3-M5 .... Mi

# Integrazione top-down – 3/4

Il processo avviene in 4 passi

- ◆ *Il main è usato come driver del test e gli stub sono costituiti da tutti i moduli direttamente subordinati. (Gli stub sono sostituiti man mano con i moduli reali)*
- ◆ *Dopo l'integrazione di ciascun modulo, si eseguono gli opportuni test*
- ◆ *Al completamento di ciascun test, un altro modulo reale viene aggiunto, sostituendolo allo stub.*
- ◆ *Può essere condotto il testing di regressione per accettarsi che non siano stati introdotti nuovi errori.*

Il processo continua dal passo 2 finché non si ottiene la completa integrazione del sistema.

Ad ogni sostituzione di uno *stub* devono essere effettuati test per verificarne l' interfaccia

# Integrazione top-down – 4/4

Problemi di natura logistica: richiedono elaborazione dei livelli bassi per poter testare in modo adeguato i livelli superiori (all' inizio ai livelli bassi vi sono ancora gli stub che potrebbero passare dati non significativi ai livelli superiori)

Tre possibilità:

1. Posporre molti test sino alla sostituzione degli stub
2. Stub più realistici
3. Integrare bottom-up

La prima provoca la perdita parziale del controllo tra test ed integrazioni

La seconda aumenta l' onere di realizzazione degli stub

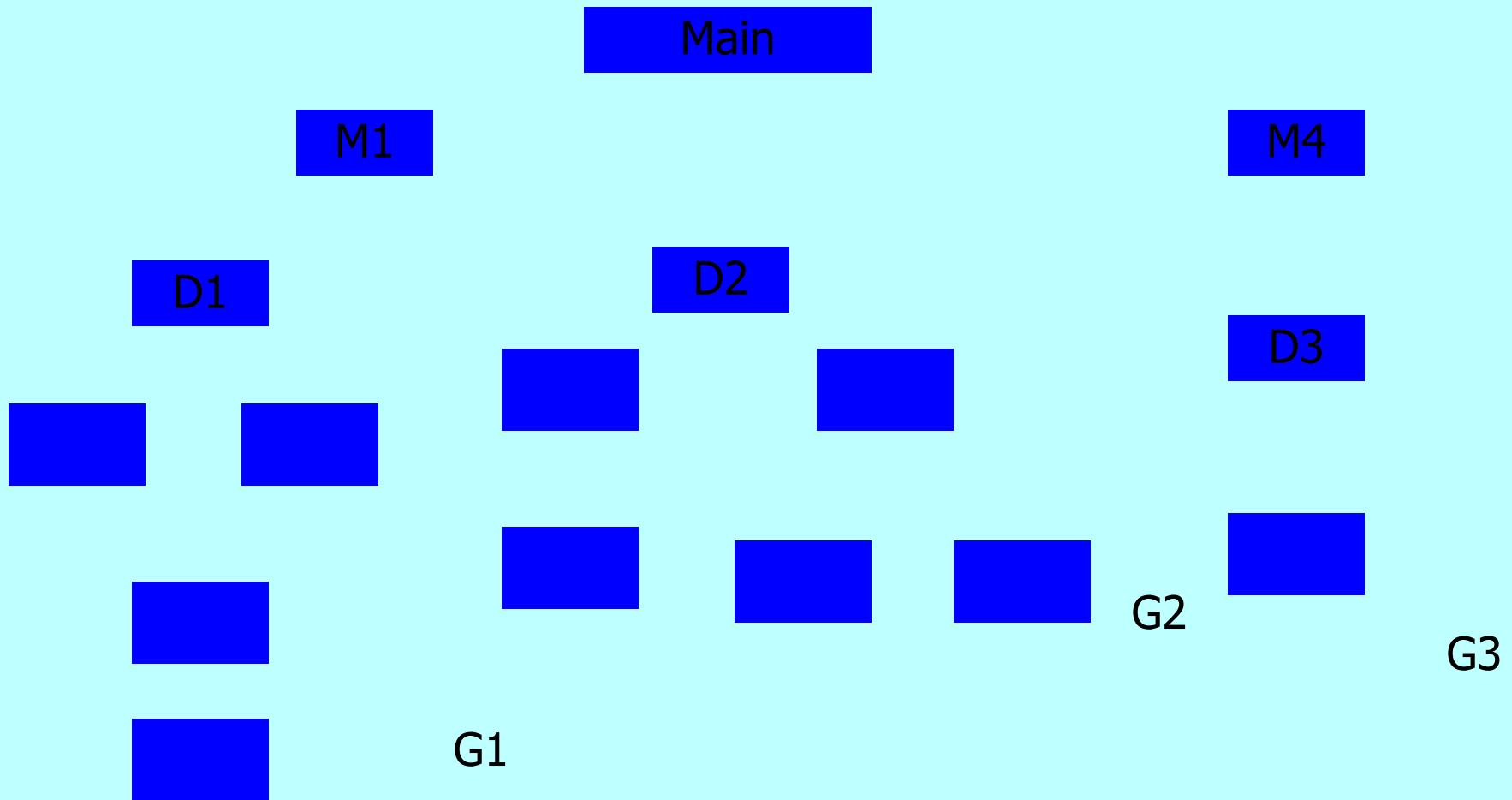
# Integrazione **bottom-up** – 1/3

L'integrazione inizia partendo dai moduli al livello più basso (quelli senza figli) e viene eliminata la necessità di stub

Strategia in più passi:

- ◆ *I moduli di basso livello vengono aggregati in gruppi (cluster) che eseguono una specifica sottofunzione software*
- ◆ *Viene realizzato un driver per coordinare l'esecuzione*
- ◆ *Si esegue il test del cluster*
- ◆ *I driver sono sostituiti dai moduli reali che vengono aggregati nel programma*

# Integrazione *bottom-up* – 2/3



# Integrazione **bottom-up** – 3/3

## ♦ Top down

Svantaggio: Stub

Vantaggio: Prova immediata delle principali funzioni di controllo

## ♦ Bottom Up

Svantaggio: Il programma non esiste fin quando non è aggregato l' ultimo modulo (il main)

Vantaggio: Test case più semplici da progettare,  
mancanza di stub, driver meno onerosi da progettare  
(per il numero ridotto di essi)

# Test di sistema

- Il software viene integrato con altri elementi del sistema (nuovo hardware, informazioni, ..) ed il tutto va testato e validato.
- Tale test non è condotto solo dall' ingegnere del software.
- Scopo del test è la verifica che tutti gli elementi del sistema siano stati correttamente integrati e svolgano bene le funzioni loro assegnate.
- **Problema dello “scarica barile”!!**

# Test di accettazione: $\alpha$ - e $\beta$ -test

Test di accettazione condotti dall' utente finale per consentire **la validazione di tutti i requisiti**

Può richiedere molto tempo e portare alla rilevazione di errori cumulativi che non sono corretti e che possono portare al degrado del sistema

Non è possibile per prodotti con molti clienti

♦  **$\alpha$ -Test**: condotto dal cliente o dall' utilizzatore finale, ma presso lo sviluppatore. Avviene in ambiente controllato

♦  **$\beta$ -Test**: condotto dall' utente finale presso uno o più clienti. Non è presente, generalmente, lo sviluppatore. Lo sviluppatore farà le opportune modifiche e poi rilascerà il prodotto

# **Processo di V&V**

# Domande fondamentali

1. Quando iniziano la verifica e la validazione?  
Quando possono dirsi **complete**?
2. Quali particolari tecniche dovrebbero essere applicate durante lo sviluppo?
3. Come possiamo stimare quando il prodotto è pronto?
4. Come possiamo controllare la qualità delle release successive?
5. Come può essere migliorato il processo di sviluppo stesso?

# 1: Quando inizia la V&V? Quando termina?

- Il test non è una fase (tantomeno l' ultima) dello sviluppo del software:
  - L' esecuzione dei test è una piccola parte del processo di verifica e validazione
- La V&V inizia non appena si decide di costruire il prodotto software, o persino prima
- La V&V dura ben oltre la consegna del prodotto, in quanto affronta i problemi derivanti dall' evoluzione del software e da adattamenti a nuove condizioni operative

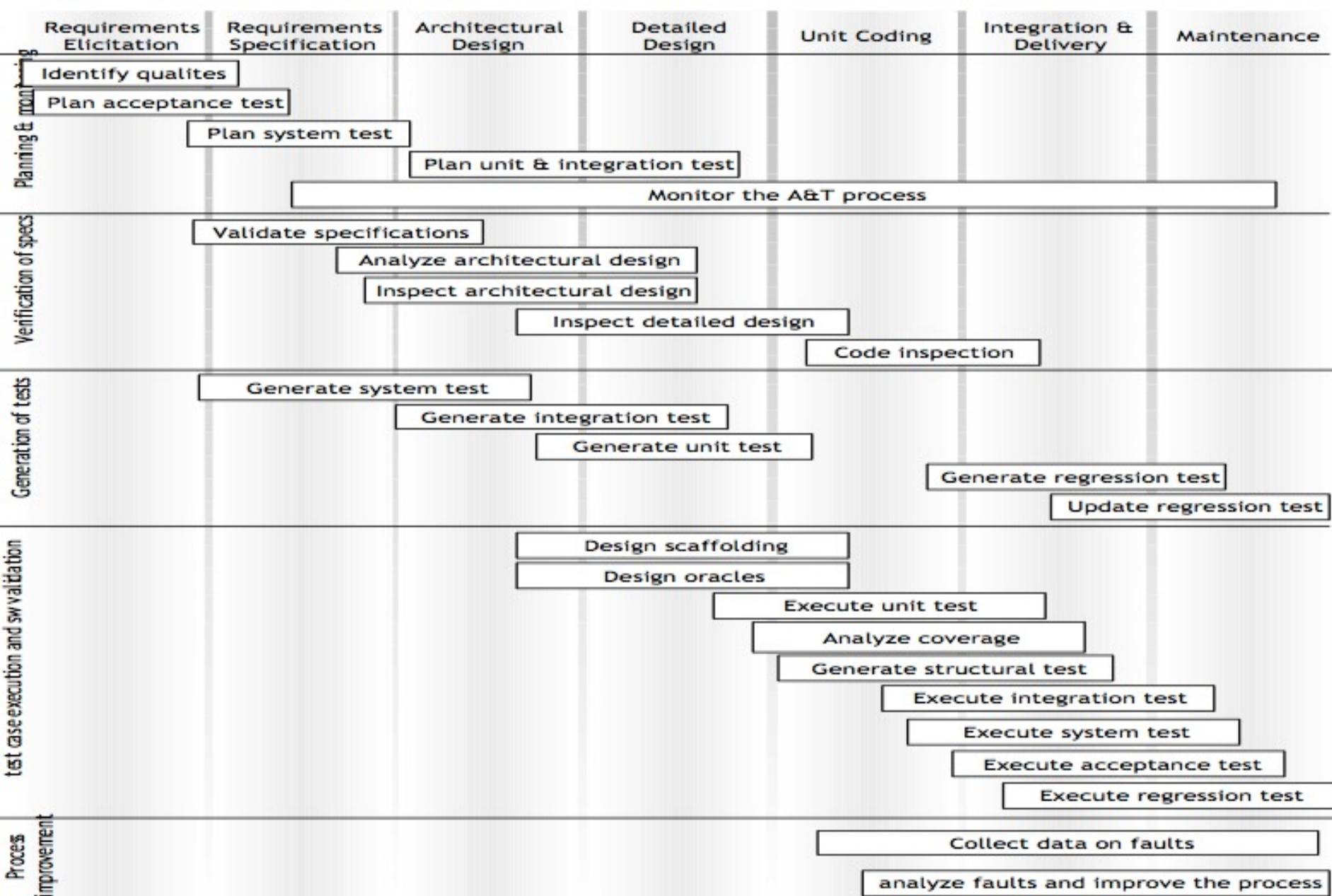
# 1: Quando inizia la V&V? Quando termina?

- ◆ In particolare, va **dallo studio di fattibilità fino alla manutenzione**, per tutta la vita del prodotto
- ◆ Nello studio di fattibilità vengono prese in considerazione le qualità richieste ed il loro impatto sul costo
- ◆ Le attività di manutenzione includono
  - Analisi dei cambiamenti ed estensioni
  - Generazione di nuove test suite
  - Ri-esecuzione di test per controllare la non regressione a valle di cambiamenti
  - Tracciamento dei difetti ed analisi

## 2: Quale tecnica dovrebbe essere applicata durante lo sviluppo?

- Non c' è una singola tecnica di Analisi e Test (A&T) che va bene per tutto. Le ragioni principali per **combinare** le tecniche sono:
  - Diversa efficienza per diverse classi di difetti: ad es., l' analisi è preferibile al test per le *race condition*
  - Applicabilità in diversi punti del progetto: ad es., ispezione per la validazione dei requisiti nelle prime fasi
  - Obiettivi diversi: ad es., lo *statistical testing* serve a misurare la *reliability*
  - Diversi tradeoff tra costo e qualità raggiunta: ad es., tecniche costose per proprietà chiave

# Organizzazione delle tecniche di A&T



### 3: Come stimare quando un prodotto è pronto?

- Le tecniche di A&T durante lo sviluppo consentono di rilevare malfunzionamenti e risalire ai difetti
- **Non è possibile rimuovere tutti i difetti**
- A&T non possono durare indefinitamente: ad un certo punto si vorrà sapere se il prodotto **soddisfa i requisiti di qualità**, per rilasciare il prodotto
- Si ha perciò bisogno di specificare il livello richiesto di qualità e determinare quando quel livello è stato raggiunto
- Alternativamente, si termina l' esecuzione delle attività di A&T prima del rilascio basandosi su altri criteri (budget massimo, tempo massimo, ecc.)

# Diversi attributi di qualità

- ◆ Affidabilità
- ◆ Robustezza
- ◆ Prestazioni
- ◆ Disponibilità
- ◆ Sicurezza
- ◆ Usabilità
- ◆ Safety
- ◆ ...

*Diverse tecniche per misurare questi attributi:*

- ◆ Tecniche di test (ad es., *statistical testing, robustness testing*)
- ◆ Tecniche di modellazione (*ad es., reliability growth models, architecture-based models, modelli basati su Markov Chains, Reti di Petri...*)
- ◆ Tecniche “measurements-based”

## 4: Come controllare la qualità delle release successive?

- Il test e l' analisi del software non terminano alla prima *release*
- I prodotti software operano spesso per molti anni e subiscono numerose modifiche
  - Si adattano ai cambiamenti dell' ambiente
  - Evolvono per soddisfare nuovi requisiti utente
- Attività orientate alla qualità dopo la consegna
  - Test e analisi di codice nuovo e/o modificato
  - Ri-esecuzione dei test di sistema
  - Raccolta e tracciamento estensivo delle informazioni

## 5: Come si può migliorare il processo di sviluppo?

- ◆ Spesso vengono riscontrati gli stessi tipi di difetti progetto dopo progetto
- ◆ Un obiettivo importante è il miglioramento delle qualità del processo, attraverso:
  - L' identificazione e la rimozione di problemi nel processo di sviluppo
  - Identificazione e rimozione di problemi nelle attività di test ed analisi, che a loro volta potrebbero rilevare problemi nel ciclo di sviluppo

# **Passi per migliorare l'analisi dei difetti e il processo**

- ◆ Definire i dati da collezionare e implementare delle procedure per collezionarli
- ◆ Analizzare i dati raccolti per identificare le classi di difetti rilevanti
- ◆ Analizzare le classi di difetti selezionate per identificare problemi nello sviluppo e nella misurazione della qualità
- ◆ Intervenire sul processo di sviluppo e sulla gestione della qualità

# Esempio di miglioramento del processo

- ◆ Si stabilisce che i difetti relativi alla sicurezza sono i più importanti
- ◆ Durante le attività di A&T si sono identificati problemi di *buffer overflow*
- ◆ Si riscontra che i difetti sono dovuti ad errate pratiche di programmazione e sono stati rilevati tardi, a causa di una scarsa attività di analisi
- ◆ Piano d' azione: modificare la disciplina e l' ambiente di programmazione, e aggiungere specifiche azioni alle *checklist* di ispezione

# Principi base di A&T

- ◆ Principi generali dell' ingegneria
  - **Partizionamento:** *divide et impera*
  - **Visibilità:** rendere l' informazione accessibile
  - **Feedback** lungo il processo di sviluppo
- ◆ Principi specifici dell' A&T:
  - **Sensitività:** meglio fallire sempre che saltuariamente
  - **Ridondanza** dei controlli
  - **Restrizione:** rendere il problema più semplice

# Proprietà del processo di qualità

- ♦ **Completezza**: sono pianificate attività appropriate per identificare classi di difetti rilevanti
- ♦ **Timeliness**: i difetti sono identificati e corretti il prima possibile
- ♦ **Cost-effectiveness**: Le attività sono scelte secondo il loro costo ed efficienza:
  - Il costo deve essere considerato sull'intero ciclo di sviluppo e su tutta la vita del prodotto

# Pianificazione e monitoraggio

- ❖ Il processo di qualità deve
  - Bilanciare diverse attività lungo il processo di sviluppo
  - Selezionare e organizzare le attività nel modo più efficiente possibile
  - Allocare le responsabilità
  - Considerare le interazioni e i tradeoff tra i vari obiettivi (ad es., *dependability* vs. *time-to-market*)
  - Migliorare la visibilità già dai primi stadi
- ❖ Gli obiettivi di qualità possono essere raggiunti solo attraverso una attenta pianificazione
- ❖ La pianificazione è parte integrante del processo di qualità

# Strategia di A&T

- ❖ Identificare gli standard rilevanti per il progetto (o per l'intera l'azienda) che devono essere soddisfatti:
  - Procedure richieste, ad es., per ottenere una certificazione
  - Tecniche e tool che devono essere usati
  - Documenti che devono essere prodotti

# Piano di A&T

- ◆ Una descrizione completa del processo di qualità che include:
  - Obiettivi e scopi delle attività di A&T
  - Documenti e altri artefatti da produrre
  - Elementi da testare
  - Caratteristiche da testare e da non testare
  - Attività di A&T e personale coinvolto
  - Criteri di accettazione (*pass or fail criteria*)
  - Pianificazione, Deliverables
  - Vincoli e requisiti hardware e software
  - Valutazione dei rischi