Ingegneria del Software

Testing del software

Prof. Paolo Giorgini

A.A. 2014/2015

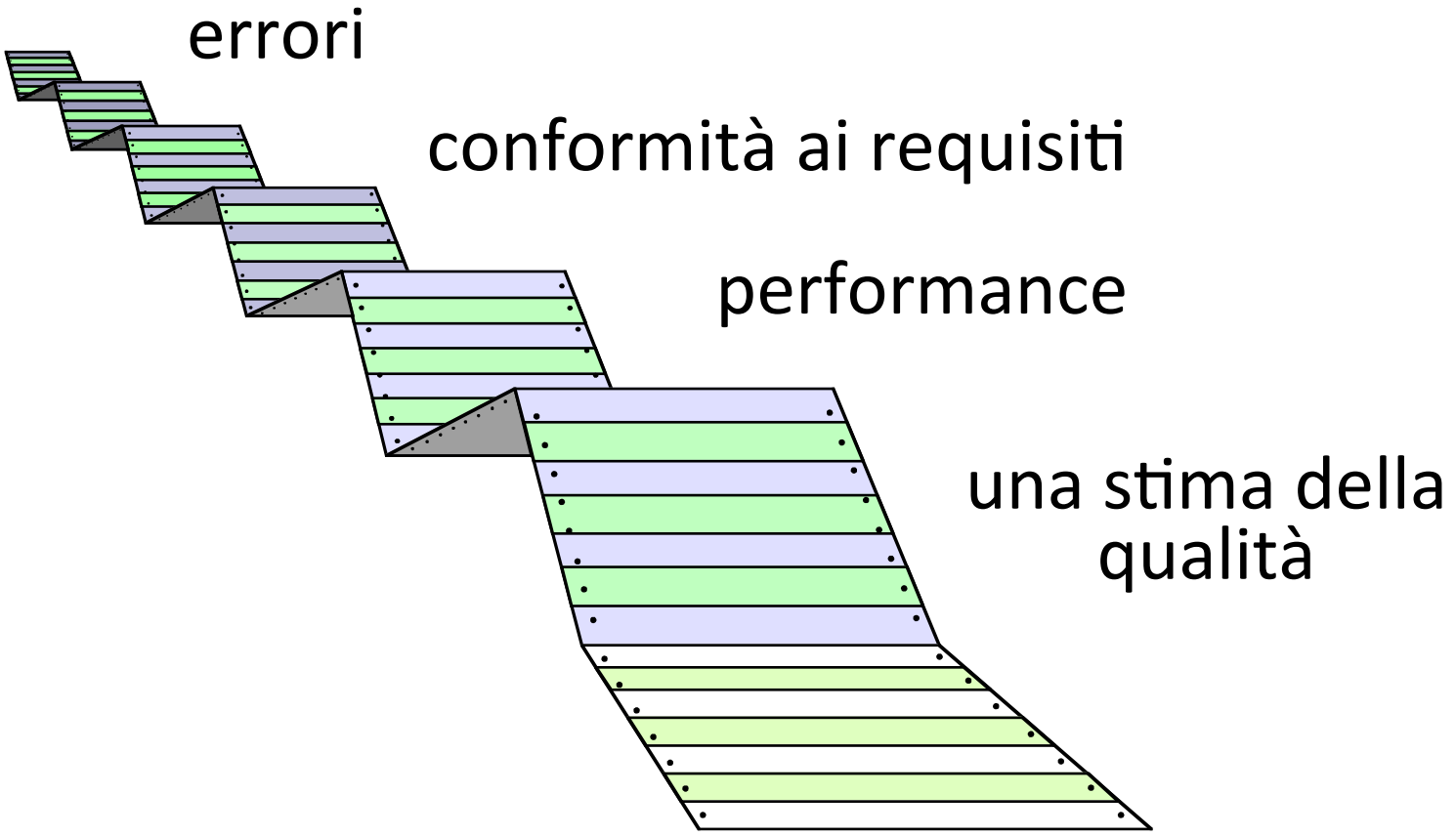
Testing del software

Il **testing** è il processo di provare il funzionamento di un programma con lo scopo specifico di individuarne gli errori prima di consegnarlo all'utente finale

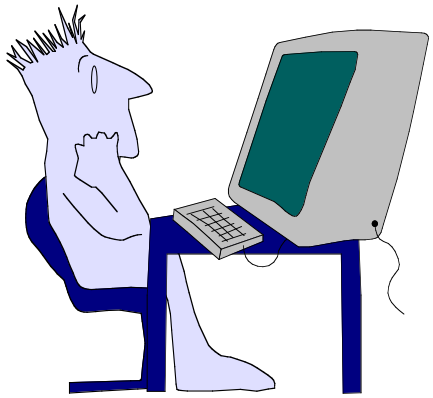
Verifica e validazione

- **Verifica**: Stiamo costruendo il prodotto correttamente?
- **Validazione**: Stiamo costruendo il prodotto corretto?

Che cosa si vede con il testing



Chi effettua il testing del software?



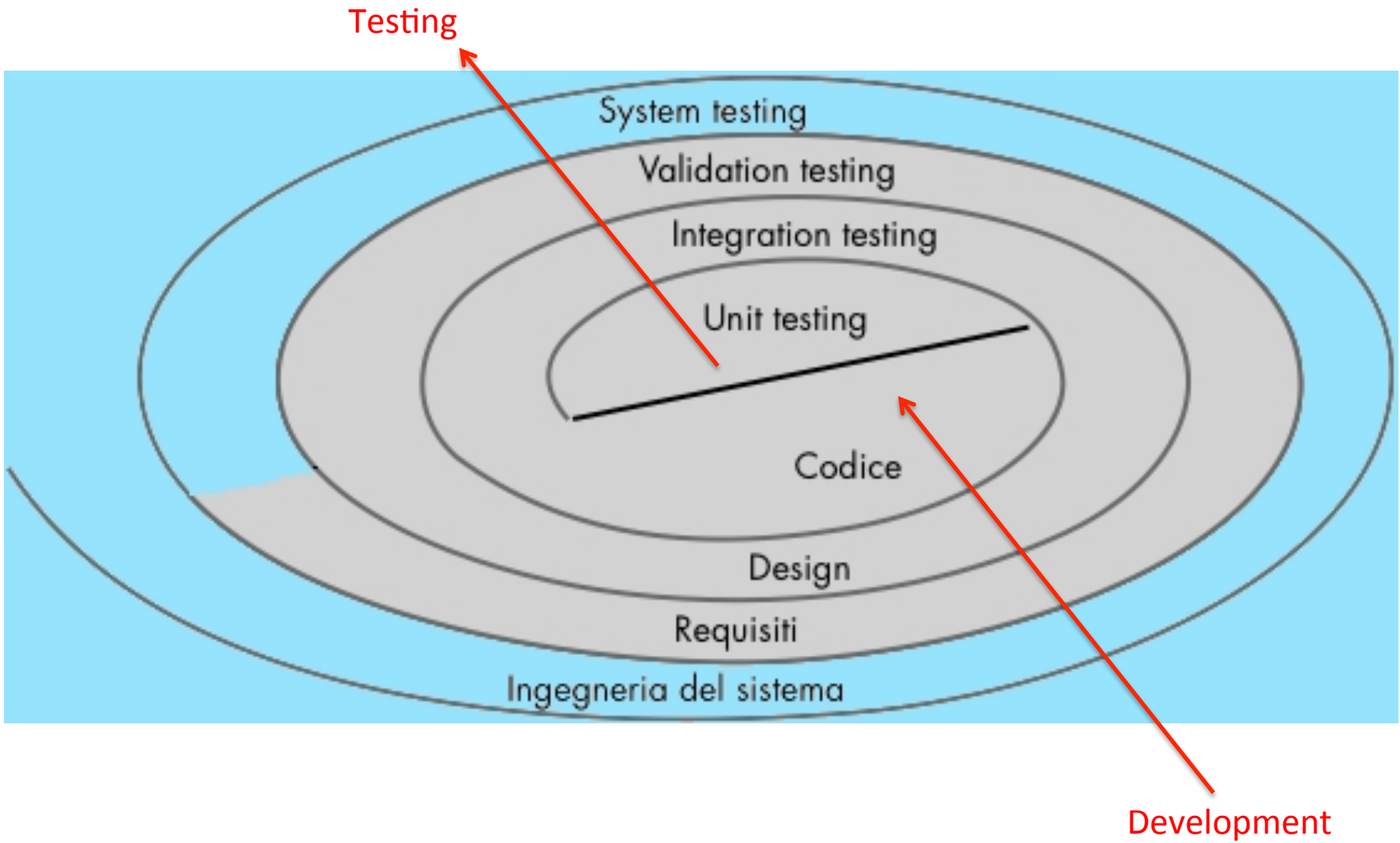
sviluppatore

Comprende a fondo il sistema ma sarà cauto nella scelta dei test e motivato dalle esigenze di "consegna"

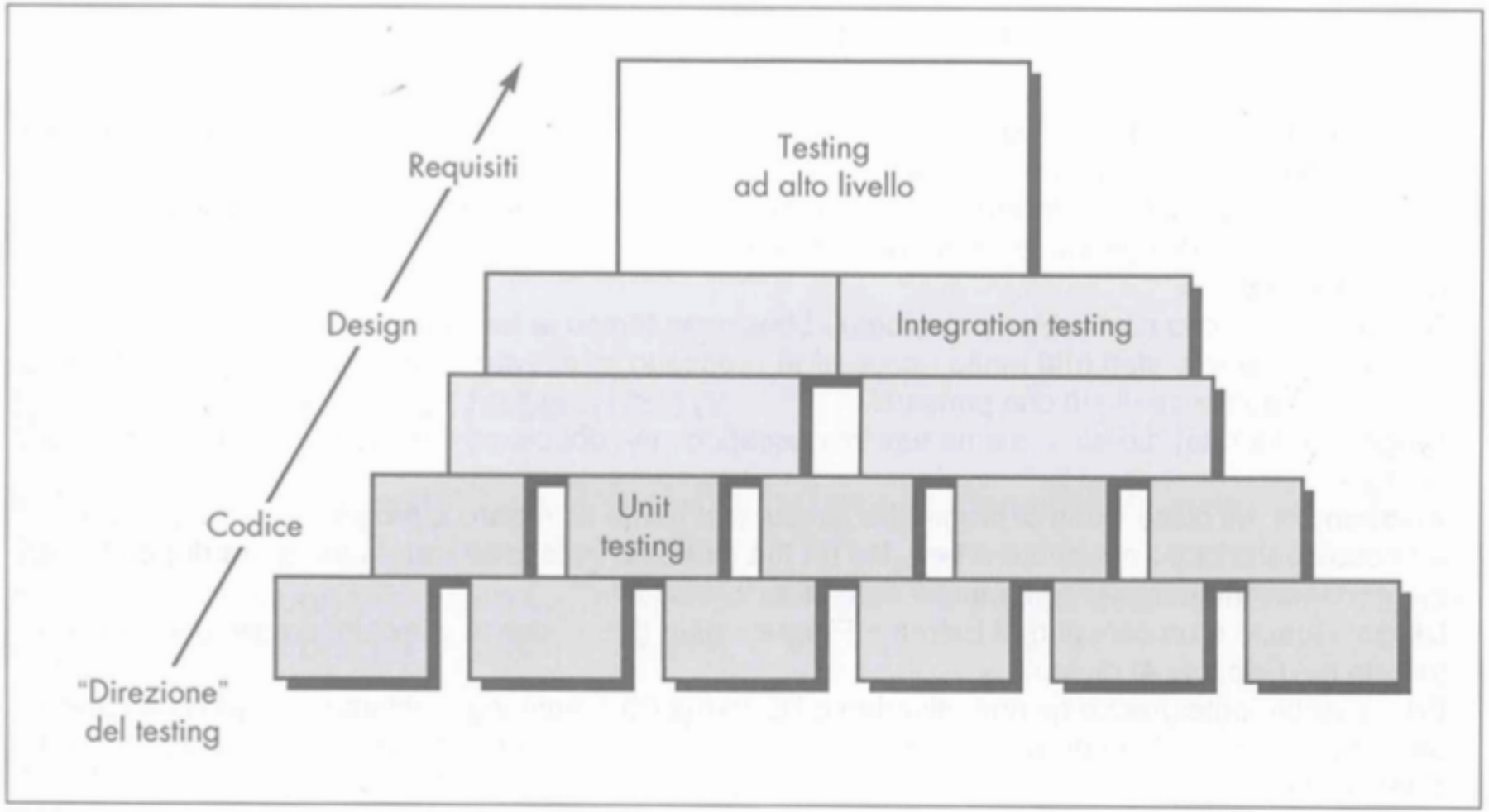
tester indipendente

Deve imparare a conoscere il sistema ma si impegnerà a trovarne i difetti e sarà guidato dalla qualità

Processo di testing



Processo di testing
(da un punto di vista procedurale)



Strategie di testing

- Si comincia con "testing in piccolo" e ci si muove verso "testing in grande"
- Nel caso del software convenzionale
  - l'attenzione iniziale è sul singolo modulo (componente)
  - in seguito l'attenzione si sposta sull'integrazione dei moduli
- Nel caso del software object-oriented
  - l'attenzione durante il "testing in piccolo" si sposta dal modulo (visione convenzionale) alla classe, che comprende attributi ed operazioni, ma anche comunicazione e collaborazione

Quando terminare il testing?

- **Il testing non finisce mai**: semplicemente la sua responsabilità passa dallo sviluppatore al cliente
  - Ogni volta che l'utente esegue un programma, è come se il programma fosse testato con una nuova serie di dati
- … oppure quando non c'è più tempo o sono finiti i soldi
- Quando è stato condotto un testing sufficiente?
  - Attraverso la modellazione statistica e la teoria dell'affidabilità del software, è possibile sviluppare dei modelli di guasto del software (individuati durante il testing) in funzione del tempo di esecuzione

Questioni strategiche (1)

- *Specificare quantitativamente i requisiti del prodotto molto prima di cominciare i test*
  - Oltre scoprire errori, si vogliono valutare altre caratteristiche legate alla qualità (es. portabilità, facilità di manutenzione e usabilità)
  - La specifica di queste caratteristiche deve essere misurabile, cos' da evitare ambiguità nei risultati del testing
- *Enunciare esplicitamente gli obiettivi del testing*
  - Espressi in termini misurabili
  - Il piano di test deve fissare l'efficacia, la copertura, il tempo medio fra guasti, il costo della scoperta e correzione, la densità residua dei difetti, la frequenza di occorrenza e la durata dei regression test
- *Sviluppare un profilo di tutte le categorie di utenti*
  - Gli use case possono ridurre il lavoro globale di testing concentrandolo sull'uso effettivo del prodotto
- *Sviluppare un piano dei test che dia risalto al "ciclo rapido"*
  - Cicli di test rapidi (2% dell'impegno di un progetto)

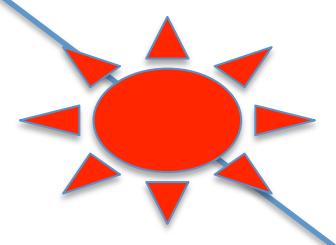
Questioni strategiche (2)

- *Costruire software "robusto", congegnato in modo da auto-verificarsi*
  - Il software deve essere in grado di auto diagnosticare certe classi di errori
  - La progettazione deve prevedere test automatici e di regressione
- *Sfruttare le revisioni tecniche formali come filtro pre-testing*
  - Molto efficaci e possono ridurre la quantità di test
- *Svolgere revisioni tecniche formali mirate a valutare la strategia di testing ed i test case*
  - permettono di rilevare incoerenze, omissioni ed errori nelle startegie di testing – risparmio di tempo
- *Seguire una via di miglioramento continuo del processo di testing*
  - Le metriche raccolte durante i test devono essere utilizzate entro una procedura di controllo statistica dei test stessi

Strategie di testing

Condurre i test sull'intero sistema

Software zeppo di errori



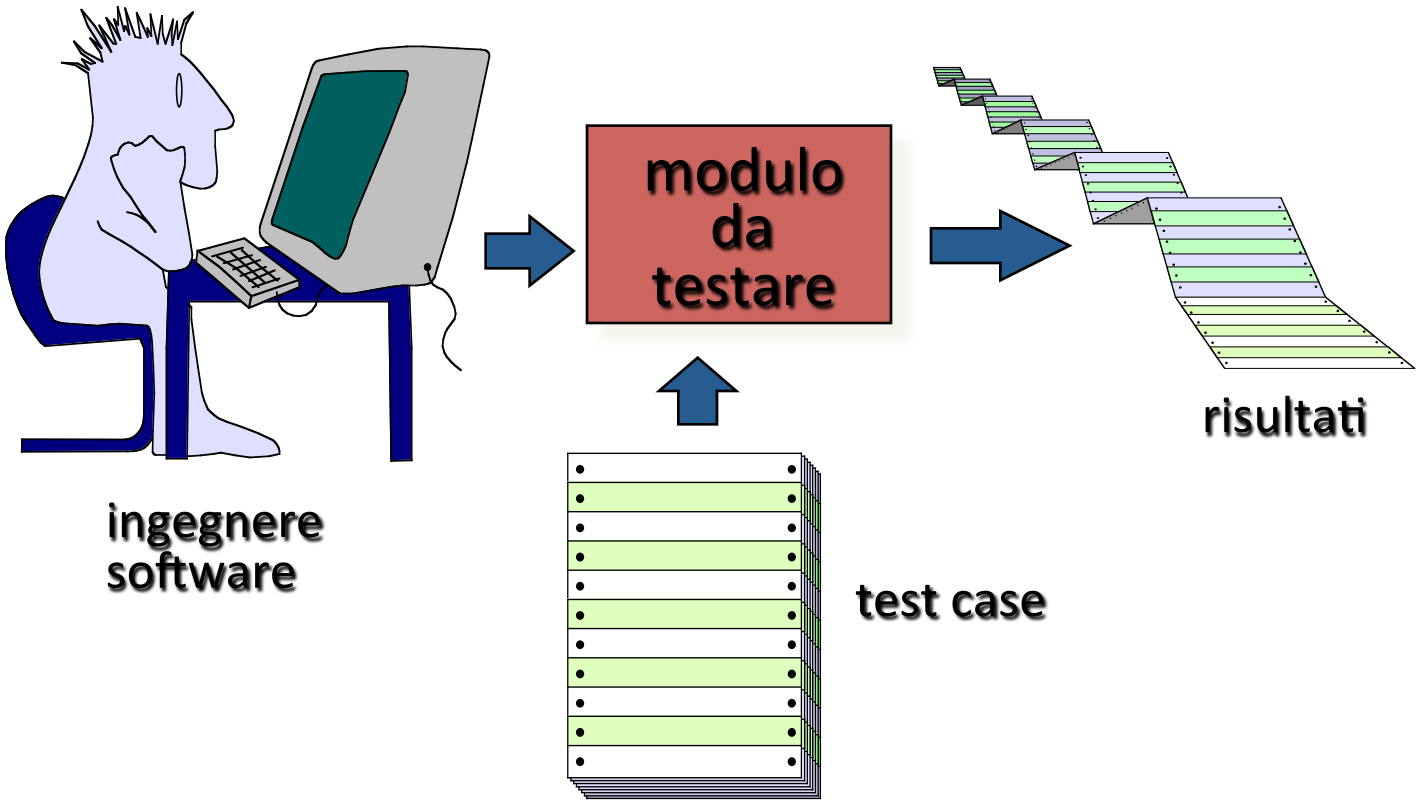
Visione incrementale del testing, iniziando con il testing delle singole unità del programma, procedendo con i test progettati per facilitare l'integrazione di queste unità e terminando con i test che mettono alla prova il sistema finale

Condurre i test quotidianamente

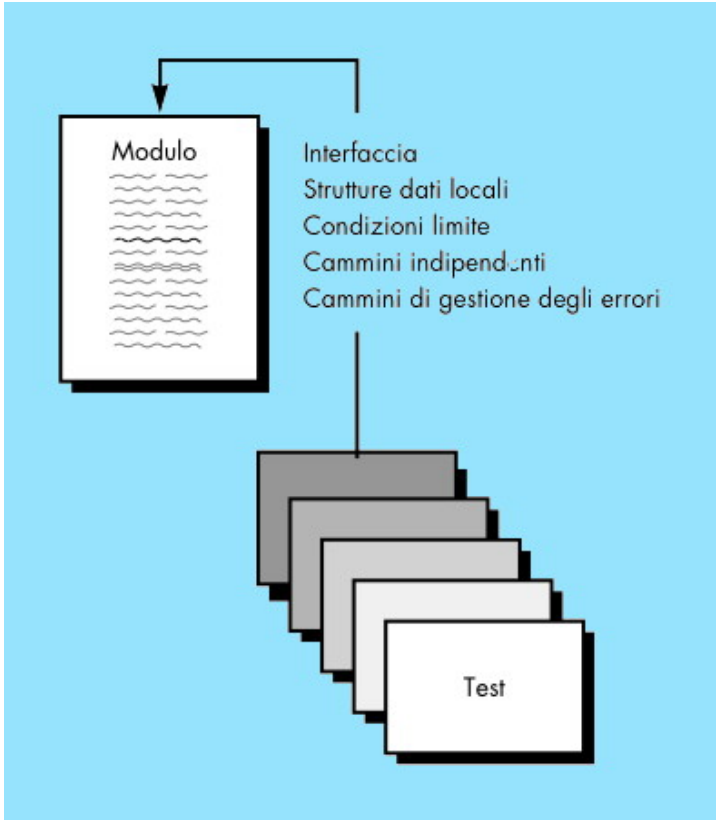
Efficace, ma gli sviluppatori non sono propensi a farlo

Unit Testing (1)

Si concentra sulla verifica della più piccola unità di progettazione software: il componente o il modulo



Unit Testing (2)



- Strutture dati locali, esaminate per assicurare che i dati memorizzati temporaneamente mantengano la propria integrità
- Cammini indipendenti della struttura di controllo vengono esercitati per assicurare che tutte le istruzioni contenute nel modulo sano eseguite almeno una volta.
- Prova di tutti i cammini per la gestione degli errori
- Prova del flusso dei dati attraverso l'interfaccia del modulo

Errori più comuni

Computazione aritmetica	Confronti e flusso di controllo
• Precedenza aritmetica errata • Uso di parametri non conforme al loro tipo di passaggio • Inizializzazione errata • Precisione insufficiente • Errata rappresentazione simbolica di un'espressione	• Confronto di dati di tipo diverso • Errori negli operatori logici o nelle precedenze • Confronto per uguaglianza quando l'imprecisione aritmetica la rende improbabile • Confronto di variabili errato • Terminazione del ciclo mancante o erronea • Mancata uscita in caso di iterazione divergente • Errore di aggiornamento delle variabili di ciclo

Boundary test e antidebugging

- Test dei casi limite (boundary test): spesso il software ha dei punti deboli in corrispondenza dei casi limite
  - L'errore si verifica quando viene eleborato *n-esimo* elemento di un array di *n* dimensioni, quando viene svolta la *i-esima* ripetizione di un ciclo di *i* passi, quando viene incontrato il valoro minimo o massimo consentito

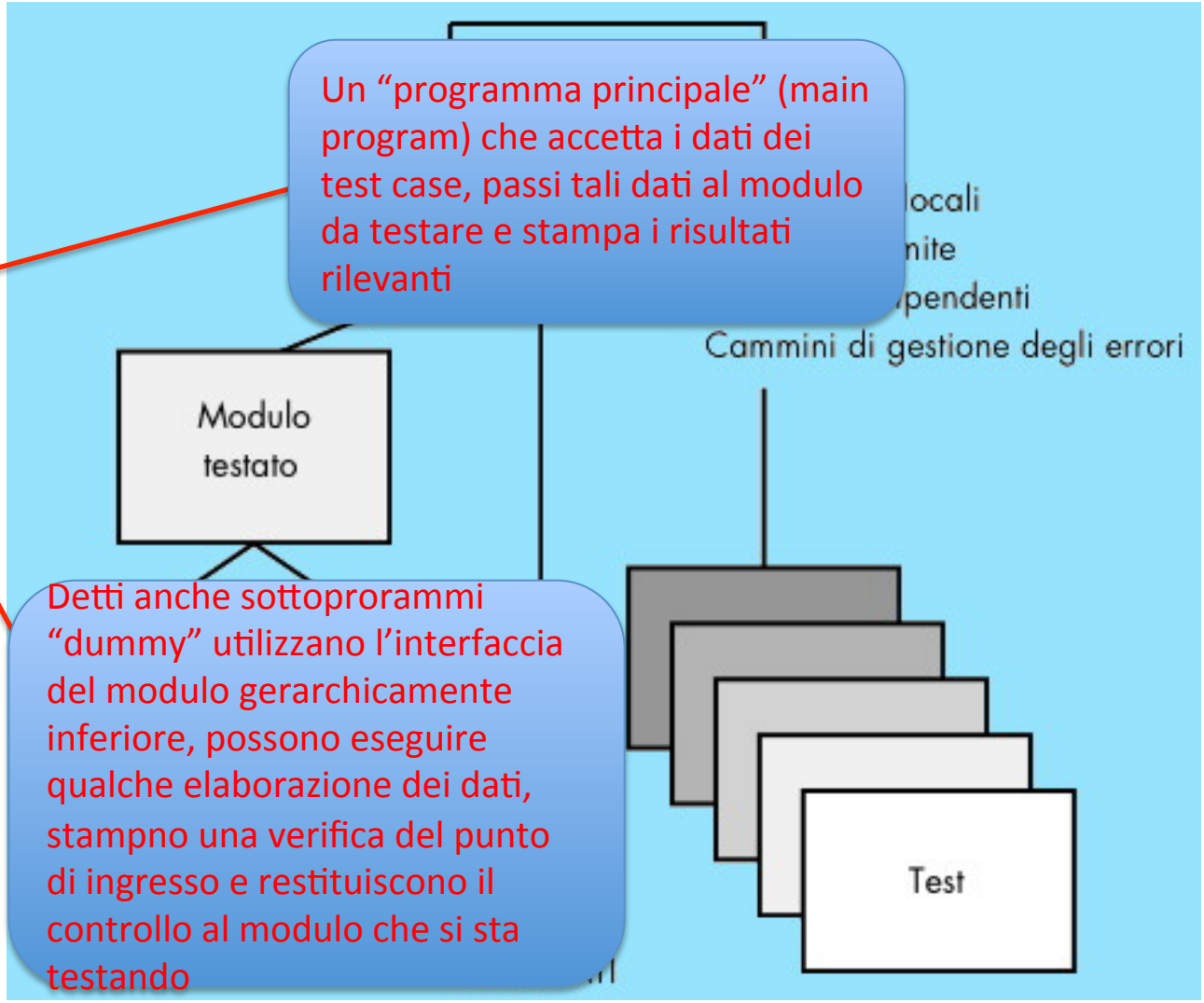
- Anticipare le condizioni di errore e impostare le sequenze di trattamento degli errori in modo da re-instradare o terminare in maniera controllata l'elaborazione dell'errore (antidebugging)

Valutazione della gestione degli errori

- Assicurarsi di progettare dei test anche per tutti i percorsi di gestione degli errori. In caso negativo un percorso di gestione degli errori potrebbe a sua volta entrare in errore.
- Errori potenziali:
  - La descrizione dell'errore è incomprensibile;
  - L'errore visualizzato non corrisponde all'errore che si è verificato;
  - La condizione di errore causa un intervento del sistema prima della gestione dell'errore stesso;
  - L'elaborazione della condizione d'eccezione non è corretta;
  - La descrizione dell'errore non fornisce informazioni sufficienti ad individuare la causa.

L'ambiente dello Unit testing

Programmi costosi da realizzare



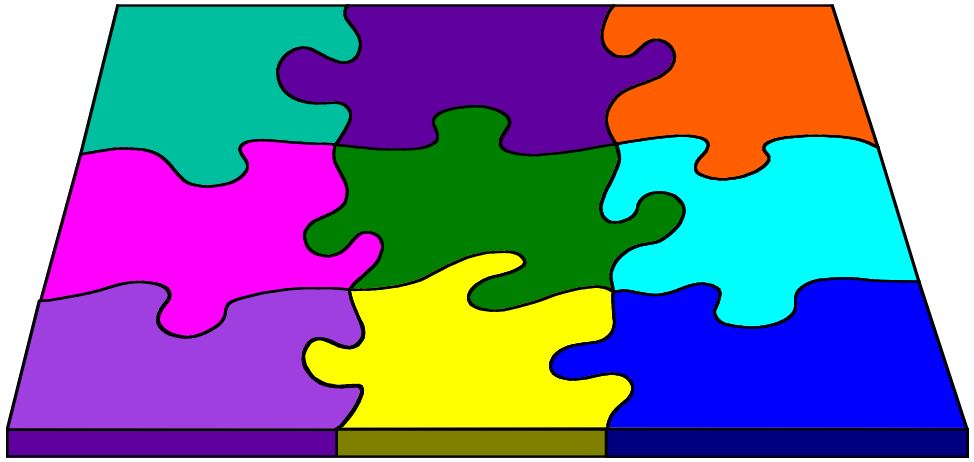
Integration testing

- Se ciascun modulo, preso singolarmente, funziona in modo corretto, perché' dovremmo dubitare sulla correttezza del loro funzionamento quando questi vengono aggregati?
  - Interfacciamento!

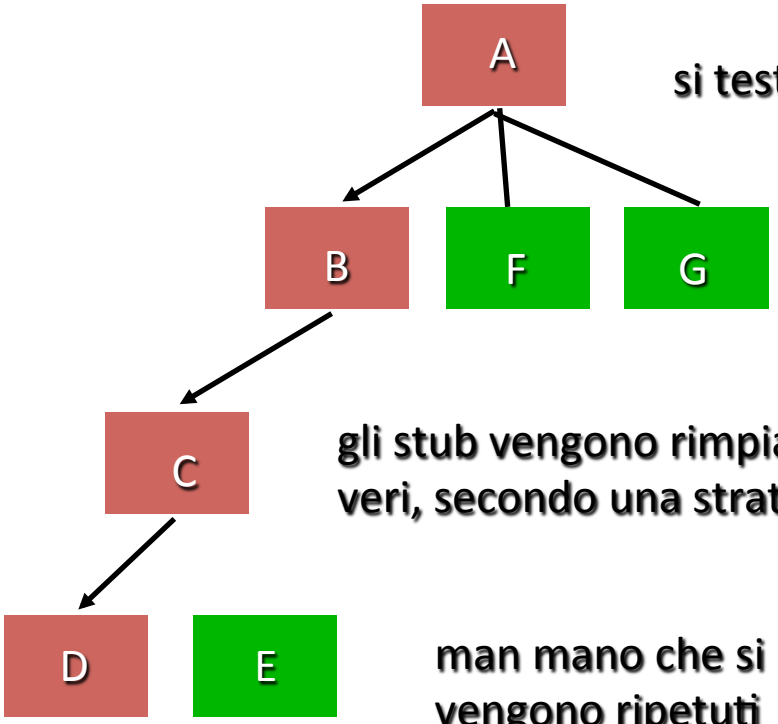
    alcuni dati possono perdersi nell'attraversare un'interfaccia; effetto negativo di un modulo su altri moduli; sotto-funzioni, combinate non sempre producono la funzione desiderata; l'imprecisione, accettabile singolarmente, può amplificarsi sino a raggiungere valori intollerabili. Strutture dati globali …

Strategie per l'integration Testing

- Opzioni
  - L'approccio a "big bang"
    - Testare il programma come sistema unitario. Fallimentare!
  - Strategie di costruzione incrementale
    - Top-down, bottom-up, regressione testing, smoke testing



Integrazione top-down



si testa il modulo top usando degli stub

gli stub vengono rimpiazzati uno alla volta dai moduli veri, secondo una strategia “depth first”

man mano che si integrano nuovi moduli alcuni test vengono ripetuti

Processo di integrazione top-down

1. Il modulo principale di controllo viene utilizzato come driver e tutti i moduli direttamente subordinati al modulo principale di controllo vengono sostituiti dagli stub
2. A seconda del modo di integrazione prescelto (deph-first o breadth-first), gli stub vengono sostituiti, uno alla volta, con i moduli reali
3. Dopo l'integrazione di ciascun modulo, si eseguono gli opportuni test
4. In seguito al completamento di ciascun insieme di test, un altro stub viene sostituito da un modulo reale
5. Può essere eseguito un regression test (vedi slide successive) per verificare che non siano stati introdotti nuovi errori

Il processo continua dal passo 2 per l'intera struttura del programma

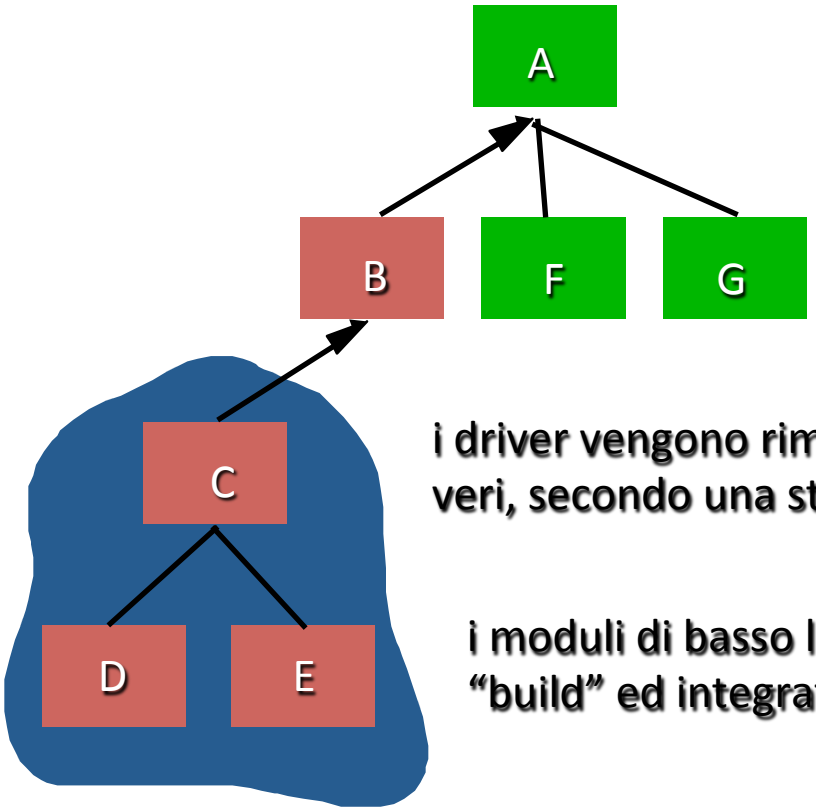
Problemi con l'approccio top-down

- Quando è richiesta l'elaborazione dei bassi livelli della gerarchia per poter provare in modo adeguato i livelli superiori
  - I moduli inferiori sono sostituiti da stub e pertanto non possono passare dati significativi a livelli superiori
    1. Rimandare molti test fino al momento in cui gli stub siano sostituiti dai moduli reali;
    2. Sviluppare stub che eseguano limitate funzioni di simulazione del modulo reale;
    3. Integrare il software dal fondo della gerarchia verso l'alto (soluzione bottom-up)

Passi dell'integrazione bottom-up

1. I componenti di basso livello vengono aggregati in cluster (gruppi detti anche build), che eseguono una determinata sotto-funzione software;
2. Viene predisposto un driver (un programma di controllo per il testing) al fine di coordinare gli ingressi e le uscite dei test case
3. Si testa il cluster
4. Si eliminano i driver e si aggregano i cluster muovendosi verso l'alto nella struttura di programma

Strategia bottom-up



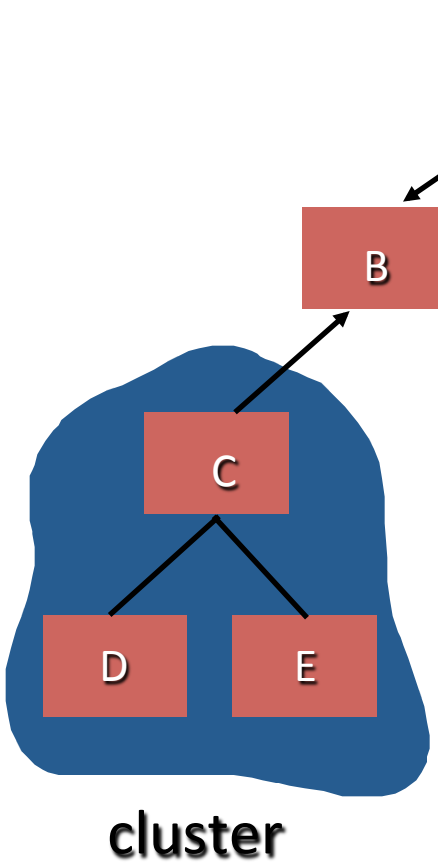
cluster

i driver vengono rimpiazzati uno alla volta dai moduli veri, secondo una strategia “depth first”

i moduli di basso livello vengono raggruppati in “build” ed integrati

si testa il modulo bottom usando dei driver

Testing Sandwich



si testano i moduli top usando degli stub

i moduli di basso livello vengono raggruppati in "build" ed integrati

Regression Testing

- Consiste nel rieseguire alcuni dei test già condotti al fine di garantire che le modifiche non abbiano prodotto effetti indesiderati
  - Riduzione dei side effect: si devono svolge dei regression test ogni volta che si apporta una grossa modifica al software (compresa l'integrazione di nuovi moduli)
  - Rilevazione errore->correzione->modifica-> regressione testing
  - Può essere fatto a mano o utilizzando tecniche di automazione
    - Test case che coinvolgono tutte le funzioni del software
    - Test case mirati alle funzioni toccate dalle modifiche
    - Test mirati ai componenti modificati

Smoke Testing

- Per prodotti software da realizzare in tempi brevi
  1. I componenti software che sono stati tradotti in codice vengono integrati in una "build"
     - include tutti i file di dati, le librerie, i moduli riutilizzabili ed i nuovi componenti necessari per implementare una o più funzionalità del prodotto
  2. Viene progettata una serie di test con lo scopo di individuare gli errori che possono impedire alla build di svolgere correttamente la propria funzione
     - Lo scopo deve essere quello di scoprire errori che hanno un'elevata probabilità di far oltrepassare la data di consegna
  3. La build viene integrata con altre build e quotidianamente viene eseguito un testing dell'intero prodotto (nella sua forma attuale)
     - L'approccio all'integrazione può essere top-down o bottom-up

Testing Object-Oriented

- Il test di una classe gioca il ruolo di unit testing
  - ogni operazione deve essere testato nel contesto della classe e delle sue sottoclassi
  - il concetto di "unit" si estende, perché ogni classe incapsula dati e metodi per manipolarli
- Le strategie di integration testing cambiano
  - nei sistemi object oriented non è chiara la gerarchia di esecuzione, quindi i concetti di integrazione bottom-up e top-down perdono di significato
  - l'integration testing basato sui thread integra le classi che devono rispondere ad un input o ad un evento
  - l'integration testing basato sui casi d'uso integra le classi che collaborano al raggiungimento di uno use case
  - l'integration testing basato sui cluster integra le classi che devono collaborare
  - la validazione si basa sui metodi convenzionali a black box
- La progettazione dei test case segue i metodi convenzionali, ma include anche tecniche specializzate

Validation testing

- Inizia dopo la fase di integration testing
  - i singoli componenti sono stati messi alla prova, il software completamente assemblato in un pacchetto e sono stati rilevati e corretti tutti gli errori di interfacciamento
- Ci si concentra sulle azioni visibili e sull'output del sistema
  - Il software funziona secondo le ragionevoli aspettative del cliente? (Specifiche dei Requisiti Software – par. sui criteri di validazione!)
  - Piano e procedura di test
    1. La funzione o la caratteristica è conforme alle specifiche
    2. Si rileva una deviazione: raramente corretti prima della release prevista, si contratta con il cliente il modo per stabilire un metodo per risolvere le mancanze

Alpha e Beta Testing

- Praticamente impossibile per uno sviluppatore prevedere come il cliente utilizzerà realmente un programma
  - Software personalizzato: serie di "acceptance test" per consentire al cliente di validare tutti i requisiti – può richiedere settimane o mesi ed è fatto dall'utente finale
  - Software generico (per molti clienti)
    - **Alpha testing**: condotto dal cliente ma presso il luogo di sviluppo. Il software viene utilizzato in maniera normale, sotto il controllo dello sviluppatore (ambiente controllato)
    - **Beta Testing**: condotto presso gli utenti finali. Lo sviluppatore non è generalmente presente (uso dal "vivo" del software).

System Testing

- Non fa parte del processo di ingegneria del software, ma possono aumentare le probabilità di successo di un progetto
- Serie di test diversi il cui scopo primario è quello di provare completamente il sistema informatico
  - Recovery testing
  - Security Testing
  - Stress Testing
  - Performance Testing

Recovery Testing

- Molti sistemi informatici devono recuperare situazioni di malfunzionamento e riprendere l'elaborazione in un tempo determinato
  - Sistemi critici, es, centrali nucleari, aeroporti, ospedali, ecc
- Tolleranza ai guasti
  - Eventuali malfunzionamenti non devono provocare l'interruzione della funzionalità complessiva del sistema
- **Recovery Testing** è un system testing che forza il software ad incorrere in un errore in molti modi diversi e verifica che il recupero sia eseguito in maniera corretta
  - **Automatico**: deve essere valutata la correttezza della reinizializzazione, dei meccanismi di salvataggio di stati a cui ripristinare il sistema in caso di un errore successivo, del recupero dei dati e del riavvio
  - **Intervento umano**: viene valutato il tempo medio per la riparazione al fine di determinare se esso rientra nei limiti accettabili

Security Testing

- Cerca di verificare che i meccanismi di protezione costruiti all'interno del sistema lo proteggano efficacemente da usi impropri
  - Chi esegue il test gioca il ruolo dell'individuo che vuole accedere al sistema: allo scopo, qualunque idea è ammessa
    - Può cercare di acquisire la password dagli impiegati, può attaccare il sistema mediante software appositamente progettato, può sovraccaricare il sistema impedendo che altri usufruiscano dei servizi, può causare di proposito errori di sistema, ecc
    - Avendo tempo e risorse un buon security testing riuscirà a violare il sistema

Regola per il progettista: fare in modo che il costo necessario per violare il sistema sia superiore al guadagno ottnuto dalle informazioni acquisite

Stress Testing

- Progettati per provare i programmi in situazioni di eccessivo carico
  - Quanto possiamo caricare il sistema prima che cada?
- Forza a richiedere risorse in quantità, frequenza o volumi eccessivi
  - possono essere progettate test speciali per generare 10 intrrupt al secondo, quando il tasso medio è 1 o 2;
  - il tasso d'arrivo dei dati d'ingresso può essere aumentato di un ordine di grandezza per verificare la risposta delle funzioni d'ingresso;
  - test case che richiedono la massima memoria possibile;
  - Test case che possono portare a saturazione la gestione della memoria virtuale;
  - Test case che provocano un'eccessiva ricerca di dati residenti sul disco

- Sensitive test: cercano di rilevare quali sono le combinazioni di dati che, pur essendo validi dati di input, possono causare instabilità od elaborazioni non appropriate

Performance Testing

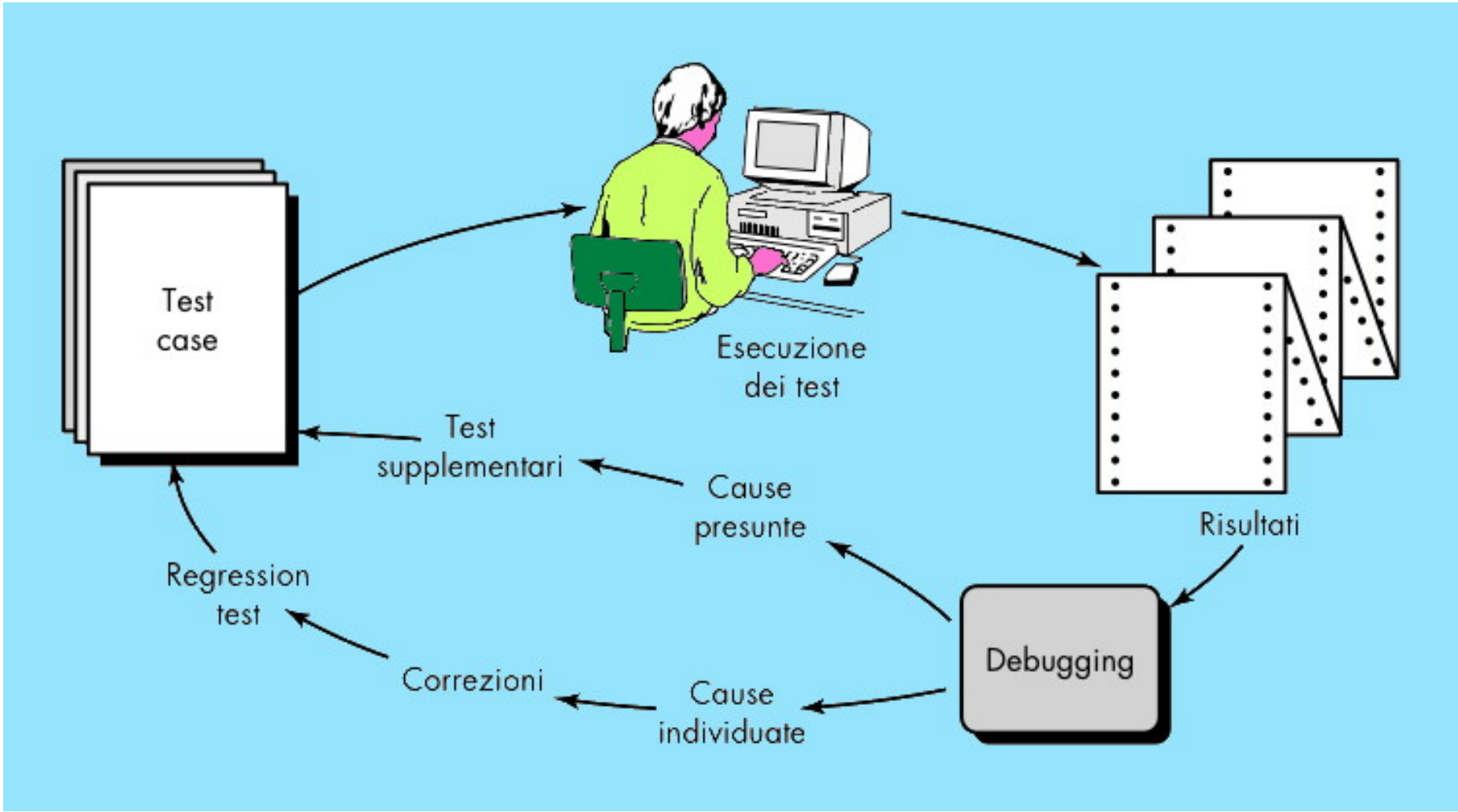
- Ha lo scopo di provare le prestazioni del software, nel contesto di un sistema integrato;
  - frequentemente associato a stress test
  - Richiedono strumentazione hardware e software ad hoc (es. per la misura precisa dei cicli di CPU)
  - Si possono rilevare situazioni che possono provocare un calo delle prestazioni oppure un malfunzionamento

Debugging: un processo di diagnosi

Quando un test case rileva un errore, il debugging è il processo che porta alla sua eliminazione

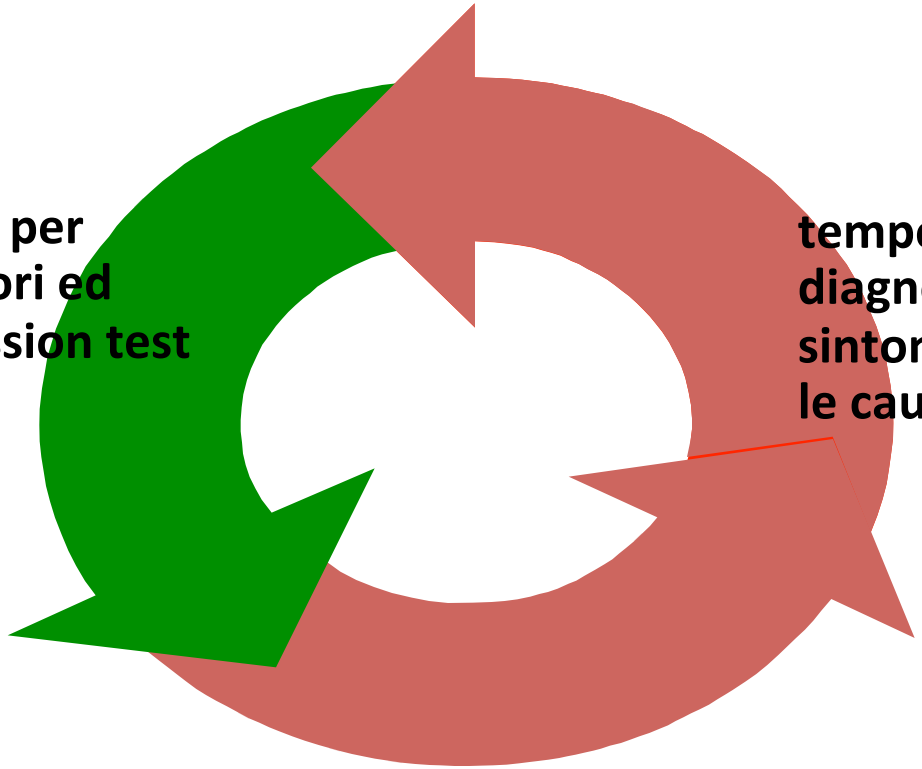
- Processo di diagnosi (ricerca delle cause dell'errore) – esperienza e conoscenza del programmatore

Il processo di Debugging



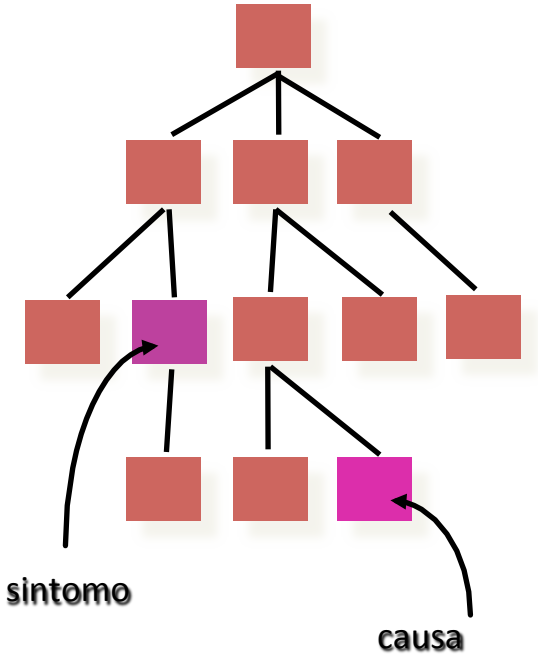
Sforzi di Debugging

tempo necessario per correggere gli errori ed effettuare i regression test



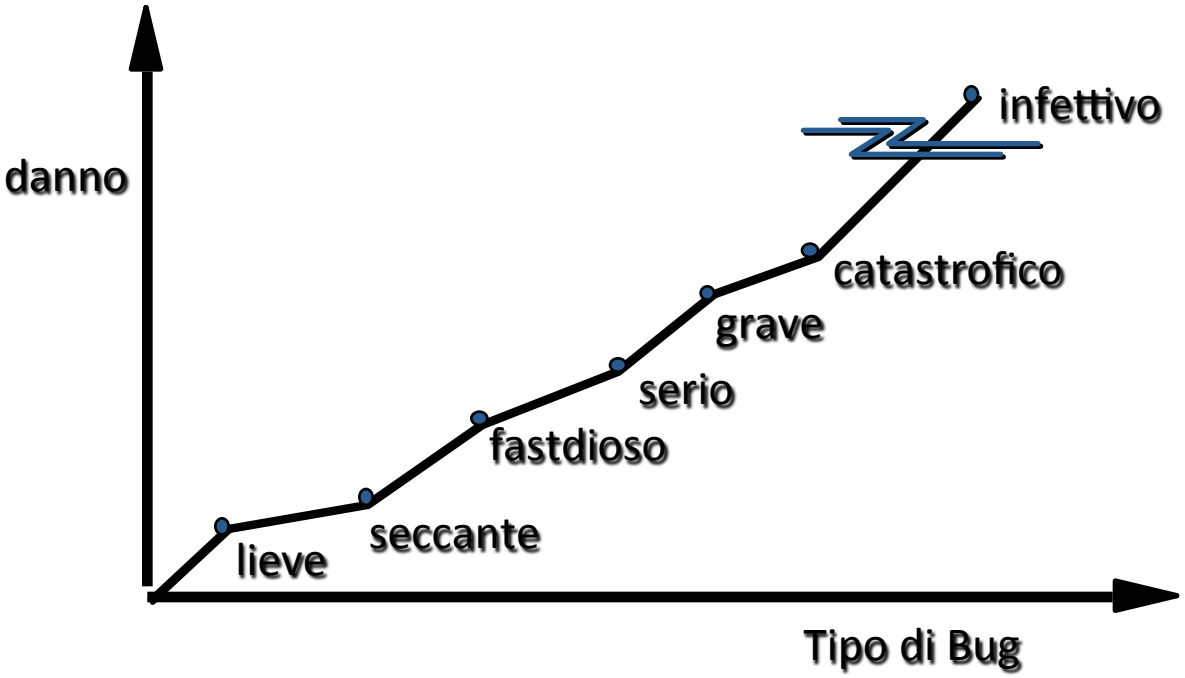
tempo necessario per diagnosticare i sintomi e determinare le cause

Sintomi e cause



- Il sintomo e la causa possono essere geograficamente lontani
- Il sintomo può scomparire solo temporaneamente per correzione di un altro errore
- Il sintomo può in realtà non essere causato da un errore specifico (per esempio da imprecisione d'arrotondamento)
- Il sintomo può essere causato da un errore umano, non facilmente individuabile
- Il sintomo può essere il risultato di problemi di temporizzazione e non di elaborazione
- Può essere difficile riprodurre accuratamente le condizioni d'ingresso precedenti all'errore
- Il sintomo può essere intermittente
- Il sintomo può derivare da cause distribuite su diversi task in esecuzione su processori diversi

Conseguenze dei Bug



- **Categorie di Bug**: bug derivanti dalla singola funzionalità, dal sistema, dai dati, dalla programmazione, dalla documentazione, da violazione degli standard, ecc.

Tecniche di Debugging

- **Forza bruta**: più diffusi e i meno efficaci; devono essere applicati quando ogni altro tentativo è fallito
  - Si raccolgono immagini della memoria e tracce dell'esecuzione, si disseminano istruzioni di scrittura nel codice (sperando di ricavare qualche indizio utile!!)
- **Backtracking:** Buoni risultati e abbastanza diffusi
  - A partire dal punto in cui si e' manifestato il sintomo, si retrocede nel codice fino ad individuare la causa (crescita dei cammini all'indietro ☹
- **Induzione/Deduzione**
  1. Si organizzano i dati legati all'errore in modo da isolare le cause potenziali
  2. Si formula un'ipotesi e si utilizzano i dati per dimostrarla o confutarla
  3. In alternativa, si compila un elenco di possibili cause e si svolgono esperimenti tesi a eliminarle una alla volta

Debugging: da non dimenticare

1. Non essere frettolosi, riflettere sui sintomi sotto esame
2. Usare strumenti (ad esempio debugger dinamici) per aumentare la propria comprensione
3. Quando non si riesce ad andare avanti, non esitare a chiedere aiuto ad altri con un nuovo punto di vista
4. Accertarsi coscienziosamente che siano stati superati tutti i regression test dopo la (presunta) correzione di un bug