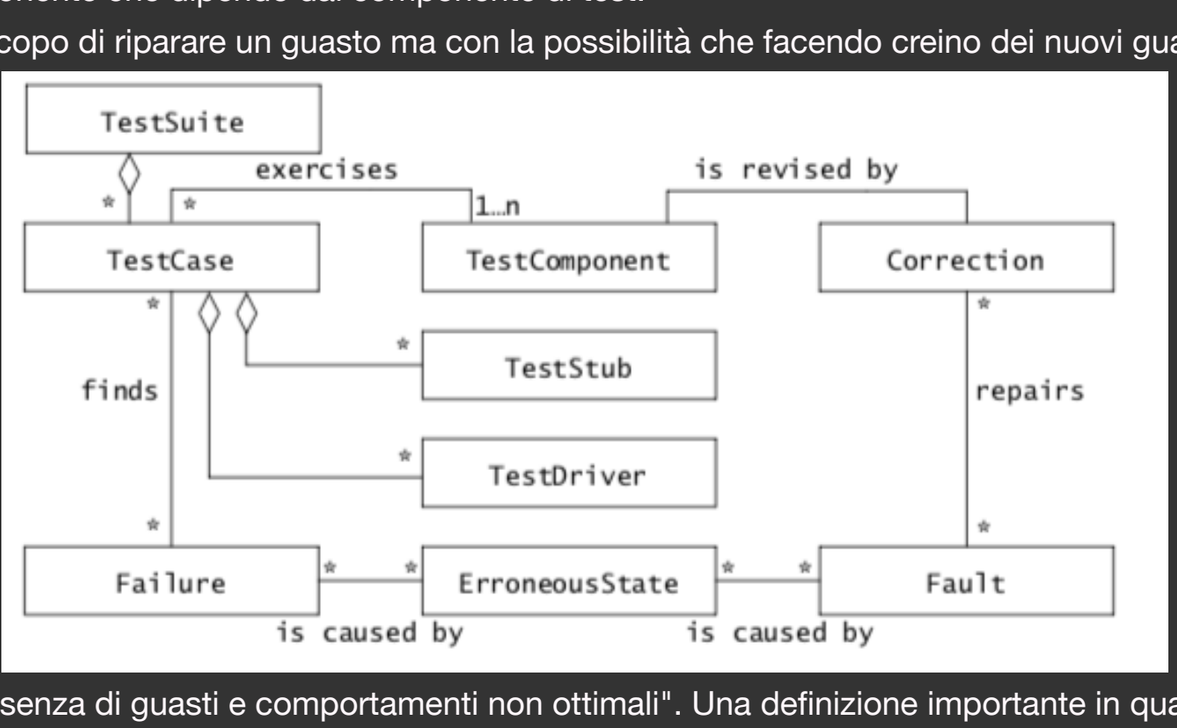
Con il testing andiamo alla ricerca delle differenze tra il comportamento previsto specificato dai modelli di sistema e il comportamento osservato del sistema implementato. Si tratta di un mero tentativo di dimostrare che l'implementazione del sistema non è coerente con i modelli di sistema. Tale fase è diversa da tutte le altre fase adottate come analisi, object design, system design in quanto questi ultimi costruiscono il sistema a differenza della fase corrispondente che cerca di rompere il sistema. Per questo motivo vengono affidati agli sviluppatori che non sono stati coinvolti nella costruzione del sistema.

Prima di procedere nel dettaglio, identifichiamo le parole chiavi che verranno utilizzate frequentemente nell'ambito Testing:

- Failure: qualsiasi devozione del comportamento osservato dal comportamento specificato.
- Erroneous state: un sistema che si trova in uno stato tale che un'ulteriore elaborazione dal parte del sistema comporterà un errore che quindi farà deviare il sistema dal comportamento previsto.
- Fault (defect o bug): causa meccanica (dipende da cause naturali) o algoritmica (dipende da cause umani) di uno stato errato.
- Test Component: una parte del sistema isolata dal resto del sistema per il testing. Un componente può essere un oggetto, un gruppo di oggetti o uno o più sottosistemi .
- Test Case: un insieme di input e risultati previsti che esercita un componente di test allo scopo di causare e rilevare guasti.
- Test Stub: un implementazione parziale dei componenti da cui dipende il componente testato.
- Test Drive: un implementazione parziale di un componente che dipende dal componente di test.
- Correction: una modifica a un componente con lo scopo di riparare un guasto ma con la possibilità che facendo creino dei nuovi guasti.

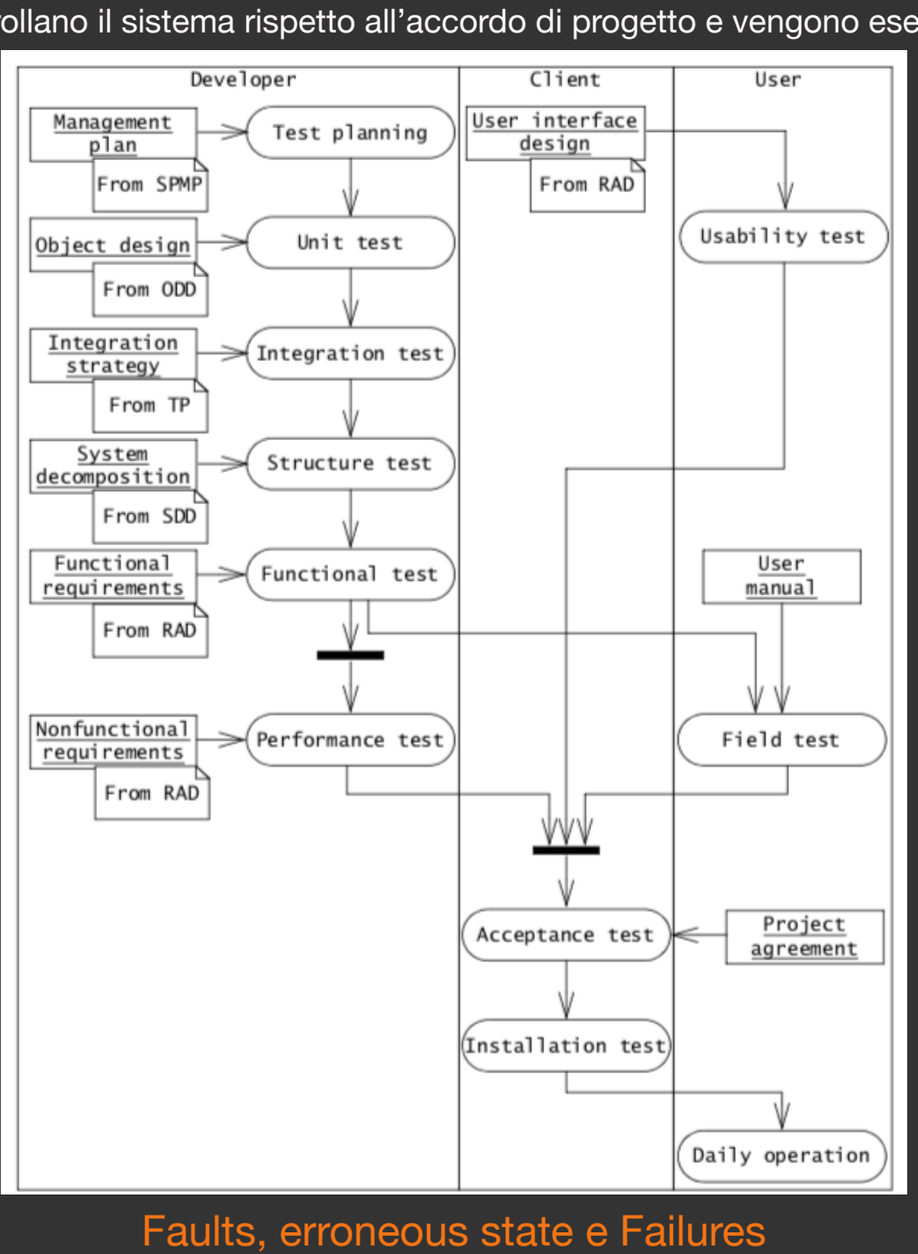


Ripetiamo, l'obiettivo esplicito del test è "dimostrare la presenza di guasti e comportamenti non ottimali". Una definizione importante in quanto ci sono degli sviluppatori che definiscono un successo se da un test non rileva nessun errore.
Se utilizziamo questa definizione: "dimostra che i guasti non sono presenti" abbiamo meno probabilità di trovare i guasti in quanto siamo spinti inconsciamente a inserire dei valori che non lo innescano. Quindi bisogna cambiare il modo di pensare rispetto alle fase costruttive; lo sviluppatore deve rilevare quanto meno i guasti nel sistema. Se nessuno dei test è stato in grado di falsificare il comportamento del sistema rispetto ai requisiti, è pronto per la consegna.
Per i test ci basiamo sull'affidabilità, una misura del successo con cui il comportamento osservato di un sistema è conforme alle specifiche del suo comportamento. Definisce anche la probabilità che un sistema non provochi guasti per un periodo di tempo specificato in condizione specifiche.
Esistono molte tecniche per aumentare l'affidabilità di un sistema software:

- Le tecniche di prevenzione dei fault:
  - si tenta di rilevare i guasti senza mettere in esecuzione il codice. L'eliminazione di tali guasti è un vantaggio in quanto impedisce che un insieme di input generino dei guasti. Tale metodo include metodologie di sviluppo, gestione della configurazione e verifica.
- Le techiche di rilevamento dei fault:
  - Con il debug e i test, si cerca di rilevare dei guasti tramite degli esperimenti incontrollati e controllati. Quindi in base agli input che genera un guasto, si determina la provenienza di tale guasto. Vengono applicare durante lo sviluppo ma in alcuni casi anche dopo il rilascio del sistema.
- Le tecniche di tolleranza dei fault:
  - Si rilascia un sistema includenti dei fault con la consapevolezza che i fault di sistema possano essere risolti recuperando da essi in fase di esecuzione. Ad esempio, i sistemi ridondanti modulari assegnano più di un componente con la stessa attività, quindi confrontano i risultati dei componenti ridondanti. Lo spacchi shuttle ha cinque computer di bordo che eseguono due diversi software per eseguire lo stesso compito.

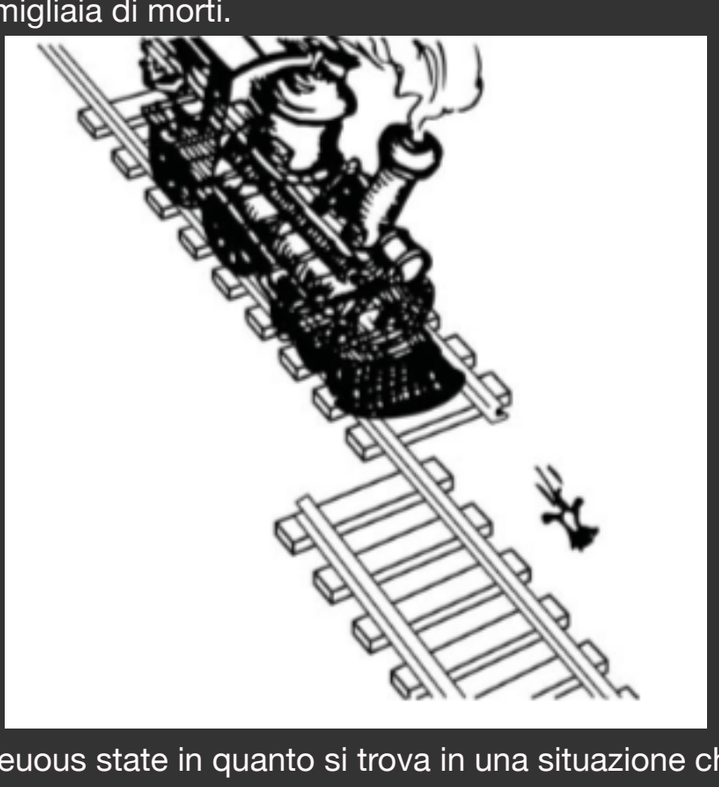
La caratteristica di un buon modello di test è che contiene casi di test che identificano i guasti in quale includono una vasta gamma din input validi, non validi e casi limite.
Il testing affronta delle seguenti attività:

- La pianificazione dei test:
  - alloca le risorse e pianifica i test. Si dovrebbe svolgere all'inizio della fase di sviluppo in modo tale che tempo e competenze siano dedicati a test. Per esempio, gli sviluppatori possono progettare casi di test non appena gli oggetti diventano stabili.
- Test di usabilità:
  - Tentativo di trovare guasti nella progettazione dell'interfaccia utente del sistema. Questo una fase importante poichè non sempre gli utenti riescono ad eseguire delle operazioni semplici per gli sviluppatori ma che risultino difficili per tali utenti.
- Unit Testing:
  - Cercare di trovare i guasti negli oggetti e/o sottosistemi isolati dal resto del sistema rispetto al caso d'uso corrispondete.
- Integration Testing:
  - Ricercare dei guasti testando i singoli componenti in "combinazione".
  - I test strutturali sono il culmine dei test di integrazione che coinvolgono tutti i componenti del sistema.
  - I due test sfruttano le conoscenze dell' SDD utilizzando una strategia di integrazione descritta nel Test Plan.
- Test di sistema:
  - Si testa tutti i componenti insieme, visiti come un unico sistema per identificare i guasti rispetto agli scenari dalla descrizione del problema e ai requisiti e agli obiettivi di progettazione identificati nell'analisi e nella progettazione del sistema:
    - Test funzionale: verifica i requisiti della RAD e del manuale dell'utente.
    - Test delle prestazioni: verificano i requisiti non funzionali e gli elementi design dell'SDD.
    - Test di accettazione e test di installazione: controllano il sistema rispetto all'accordo di progetto e vengono eseguiti dal client con l'aiuto degli sviluppatori.



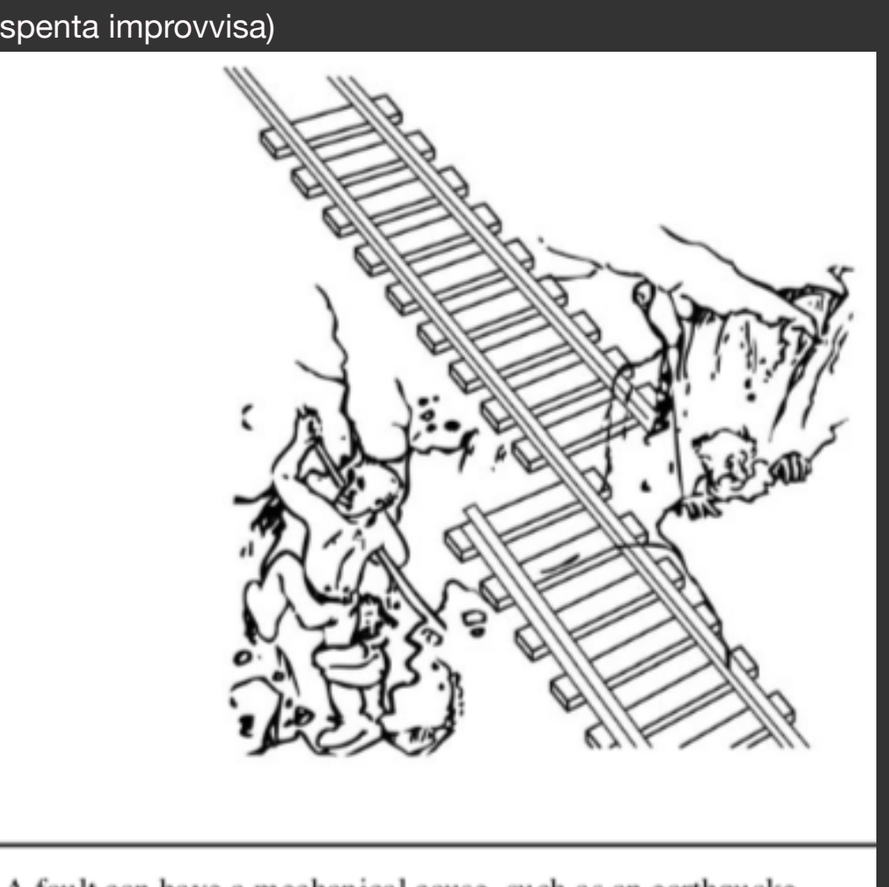
Faults, erroneous state e Failures

Con questa immagine non riusciamo a determinare se sia un fault, un erroneo state o un failers in quanto non è stato specificato nessun comportamento previsto.
Quindi per individuare di che tipo di errore sia, è necessario confrontare il comportamento desiderato (descritto nel caso d'uso nella RAD) con il comportamento osservato (descritto dal caso di prova).
Supponiamo di avviare un caso d'uso con un treno che si sposta dalla pista in alto a sinistra alla pista in basso a destra. Con questo possiamo dimostrare che il sistema contiene un fault poichè con dei binari fuori posto, il treno si capovolterà causando così migliaia e migliaia di morti.



Inoltre con questa immagine possiamo dimostrare che si tratta di un erroneuous state in quanto si trova in una situazione che innescherà un fauilre.
I disallineamento delle tracce può essere il risultato di una cattiva comunicazione tra i team di sviluppo. Tale esempio corrisponde ad un fault algoritmico poichè un fault causato dagli umani. A differenza del fault meccanico innescato da eventi naturali come un terremoto che ha provocato il disallineamento dei binari oppure il failure in un componente del sistema di alimentazione che ha portato ad un fault meccanico in un altro componente del PC causando così il failure di quest'ultimo (spenta improvvisa)

A fault can have an algorithmic cause.



A fault can have a mechanical cause, such as an earthquake.

Test Cases

Un caso di test è un insieme di dati di input e risultati previsti che esercita un componente allo scopo di causare e rilevare guasti.
Un caso di test ha cinque attributi:

1. Nome:
   - Consente al tester di distinguere tra diversi casi di test. Si consiglia quindi di rinominare tale test con il termine del metodo che lo si testando. Esempio metodo getCont()—> nel testing si chiamerà Test_getCont().
2. Posizione
   - Descrive dove è possibile trovare il caso di test. Può essere il nome del percorso o l'URL dell'eseguibile del programma di test e dei suoi input.
3. input:
   - Descrive l'insieme di dati di input o comandi che devono essere immessi dall'attore del test case.
4. Oracolo:
   - Descrive il comportamento previsto di tale test case.
5. Registro:
   - Un insieme di correlazioni con data e ora del comportamento osservato con il comportamento previsto per varie esecuzioni di test.

Una volta che i test sono identificati e descritti si determinano le relazioni tra questi:

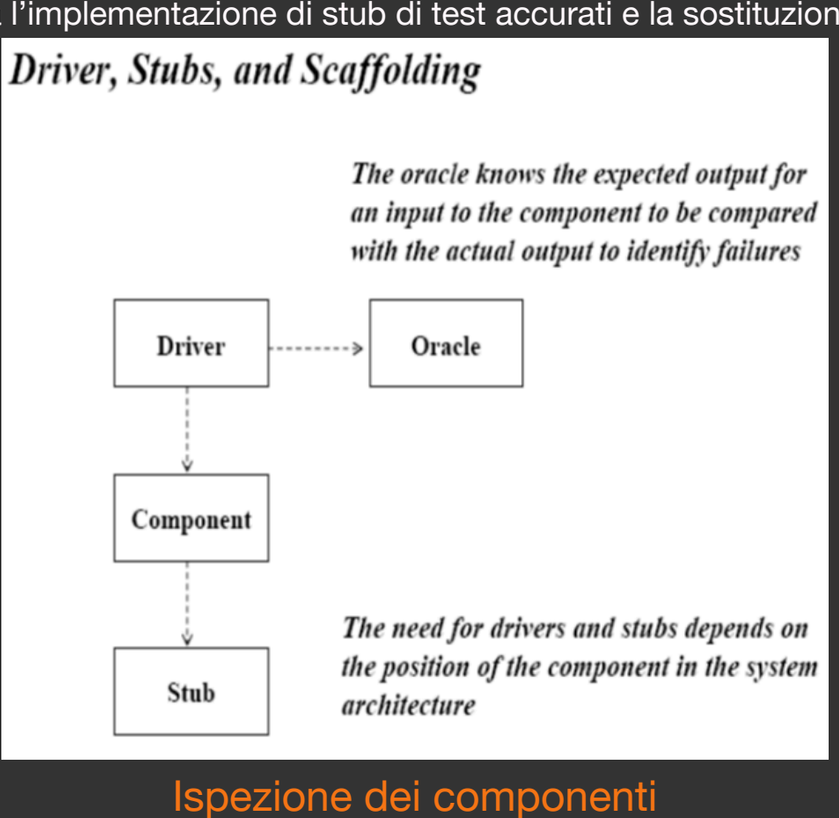
- Aggregation:
  - Usata quando un test case può essere decomposto in un insieme di subtest.
- Precedence:
  - 2 test case sono caratterizzati da questa relazione quando un test case deve precedere un altro test case.

Si cerca di aver un modello di test con poche relazioni in modo da velocizzare il processo di testing.
I casi di test sono classificati in:

- Test Blackbox:
  - Non vede la struttura interna del sistema ma si concentra sul comportamento di input/output del componente.
- Test Whitebox:
  - Si concentra sulla strutta interna del componente assicurando che venga testato ogni stato nel modello dinamico degli oggetti e ogni interazioni tra gli oggetti.

Test Stub e Driver

Eseguire dei casi di test su singoli componenti o sottosistemi richiede che essi siano isolati dal resto del sistema. Di conseguenza i test driver e i test sub verranno utilizzati per sopperire alle parti mancanti del sistema.
Un test drive simula la parte del sistema che andrà a chiamare il componente testato passando gli input dove produrrà a sua volta i risultati.
Un test stub simula un componente chiamato dal componente testato. Si tratta di un'attività non tanto banale rispetto al test drive in quanto non è sufficiente creare un componente stub che stampi semplicemente un messaggio ma è necessario che tale componente simuli esattamente il comportamento dell'oggetto mancante. Perché se così non fosse, potrebbe generare un failure non a causa del componete testato ma a causa del componente stub. Nemmeno fornire un valore di ritorno è sufficiente poichè potrebbe ritornare sempre lo stesso valore, un evento non adatto per un determinato scenario. Di conseguenze esiste un compresso tra l'implementazione di stub di test accurati e la sostituzione degli stub di test con il componente effettivo.



Ispezione dei componenti

Una volta convalidato un oggetto, si provvede ad ispezionare il codice sorgente di tale componente. Le ispezioni possono essere condotte prima o dopo il test di unità.
Il primo processo di ispezione fu invitato da Fagan.
E' formato da un team di sviluppatori, tra cui l'autore del componente, un moderato che facilita il processo e uno o più revisori che trovano guasti nel componente
Fagan prevede 5 passaggi:

1. Panoramica: l'autore del componente presenta brevemente lo scopo e la portata del componente e gli obiettivi dell'ispezione.
2. Preparazione: i revisori acquisiscono familiarità con l'implementazione del componente.
3. Riunione di ispezione: un lettore parafrasa il codice sorgente del componete e il team di ispezione solleva problemi con il componente. Un moderatore mantiene la riunione sulla buona strada.
4. Rielaborazione: l'autore rivede il componente.
5. Azione supplementare: il moderatore controlla la qualità della rilavorazione e può determinare il componente che deve essere riesaminato.

Purtroppo i passaggi definiti da Fagan sono da considerati troppo dispendioso in termini di tempo a causa della lunghezza della fase di preparazione e di ispezione. Quindi Parnas ha riformulato tale processo in nuovo processo. Si tratta di ispezione riveduto, una revisione attiva del progetto, la quale elimina la riunione di ispezione di tutti i membri del team di ispezione. Tuttavia ai revisori viene chiesto di trovare i guasti durante la fase di preparazione. Al termine della fase di preparazione, ciascun revisore compila un questionario che verifica la propria comprensione del componente. Dopodiché l'autore incontra individualmente ogni revisore per raccogliere dei feedback sul componente ispezionato.
Sia le ispezione di Fagan che il processo di Parnas sono rivelati più efficaci dei test per scoprire i guasti. Sia i test che le ispezioni vengono utilizzati in progetti critici per la sicurezza, poichè tendono a trovare diversi tipi di guasti.

Correzioni

Se in un test vengono rilevati dei guasti, gli sviluppatori provvedono ad correggere tail fault. Tali correzione possono variare da una semplice modifica a un singolo componente, a una riprogettazione completa di una struttura dati o di uno sottosistema aumentando al contempo la probabilità di introdurre dei nuovi guasti.
Vi sono diverse tecniche a fin di garantire un minor probabilità di innescare dei nuovi guasti:

- Il rilevamento dei problemi:
  - Include la documentazione di ogni failure, erroneous state e fault rilevato, la sua correzione e le revisioni dei componenti coinvolti nella modifica. Consente agli sviluppatori di restringere la ricerca di nuovi guasti.
- I test di regressione:
  - Dopo aver corretto un fault, si procede ad rieseguire tutti i test in modo da garantire che tale correzione non abbia introdotto dei nuovi guasti. Il test di regressione è importante nei metodi orientati agli oggetti, che richiedono un processo di sviluppo iterativo. Ciò richiede che i test vengano avviati prima e che i test vengano mantenute dopo ogni iterazione. Purtroppo sono costosi.
- La manutenzione razionale:
  - Include la documentazione della logica del cambiamento e il suo rapporto con la logica del componente rivisto. Consente agli sviluppatori di evitare l'introduzione di nuovi guasti ispezionando i presupposti utilizzati per costruire il componente.

Test di usabilità

Il test di usabilità verifica la comprensione del sistema da parte dell'utente. Non viene fatto nessun confronto con un sistema specifico bensi' si cerca di scoprire le differenze tra il sistema utilizzato e le aspettative degli utenti su ciò che dovrebbe fare. Per far ciò, si utilizza un "approccio empirico": gli sviluppatori formulano una serie di obiettivi di test descrivendo ciò che desiderano di imparare dal test. Dopodiché i partecipanti trovano problemi manipolando l'interfaccia utente del sistema o una sua simulazione. In questo modo gli sviluppatori sono in grado di raccogliere delle informazioni importanti quali il layout geometrico delle varie componente, l'aspetto dell'interfaccia utente. Ma anche informazioni riguardanti le prestazioni degli utenti e preferenze per identificare problemi specifici con il sistema o raccogliere idee per migliorarlo.
Esistono tre tipi di test di usabilità:

- Test dello scenario:
  - Durante questo test, a uno o più utenti viene presentato uno "scenario visionario" del sistema. Quindi gli sviluppatori osservano con che rapidità sono in grado di comprendere lo scenario ed in che modo reagiscono positivamente alla descrizione di tale scenario. Tali scenari dovrebbero essere più realistici e dettagliati possibili. Permette agli sviluppatori di ottenere un feedback rapido da parte dell'utente.
  - Vantaggio:
    - Sono economici da produrre in quanto vengono svolti in cartaceo.
  - Svantaggio:
    - L'utente non può manipolare il sistema.
- Test Prototipo:
  - Durante questo test, vengono realizzati dei prototipi che implementano degli aspetti chiavi. Quindi gli utenti interagisco con tale prototipo. Ci sono due tipi di prototipi:
    - Prototipo verticale: implementa completamente un caso d'uso del sistema. Vengono utilizzati per valutare i requisiti fondamentali come il tempo di risposta del sistema o il comportamento dell'utente sotto stress.
    - Prototipo orizzontale: implementa un singolo strato del sistema come l'interfaccia utente senza fornire funzionalità. Vengono utilizzati per scoprire problemi ricorrenti all'interfaccia utente o layout delle finestre.
  - Vantaggio:
    - Forniscono all'utente una visione realistica del sistema. Ma non solo, permette anche il raccoglimento dei dati dettagliati.
  - Svantaggio:
    - Richiedono uno sforzo maggiore per la costruzione rispetto al testo dello scenario.
- Test Prodotto:
  - Simile al test di prototipo tranne per il fatto che al posto del prototipo viene utilizzata una versione funzionale del sistema. Può essere condotto solo dopo lo sviluppo della maggior parte del sistema. Inoltre richiede che il sistema sia facilmente modificabile.

In tutti e tre tipi di test, vi sono degli elementi di basi:

- Sviluppo di obiettivi di test;
- Un campione di utenti finali;
- L'ambiente di lavoro reale o simulato.
- L'interrogazione estesa e controllata e sondaggio degli utenti da parte della persona che esegue il test di usabilità
- Raccolta e analisi dei risultati quantitativi e qualitativi
- Raccomandazioni su come migliorare il sistema.

Unit Testing

Si concentrato ad eseguire i test in merito a un singolo componete o un sottosistema.
Vi sono dei motivi per cui è efficiente:

1. Riducono la complessità della attività di test pesanti, consentendoci di focalizzarsi su problemi più piccoli.
2. Semplificano l'individuazione e la correzione dei guasti, dato sono isolati dal resto del sistema.
3. Consente il parallelismo: ogni componete può essere testato indipendentemente dagli altri.

Gli oggetti da testare dovrebbero essere quei partecipanti nei casi d'uso. Inoltre i sottosistemi dovrebbero essere testati componenti solo dopo che ciascuna delle classi all'interno di quel sottosistema è stata testata individualmente. Sottosistemi esistenti, che sono stati riutilizzati o acquistati, devono essere tratti come componenti con struttura interna sconosciuta.
In seguito diversi tipi di unit testing per la blackbox:

- Test di equivalenza
- Boundary Testing

Per la whitebox vengono eseguiti diversi tipi di testing tra cui:

- Statement Testing
  - Si testano i singoli statement
- Loop Testing
  - Definisce i casi di test in modo da assicurarsi che:
    - Il loop che deve essere saltato completamente
    - Loop da eseguire almeno una volta
    - Loop da eseguire più di una volta.
- Path Testing
  - Assicura che tutti i percorsi siano eseguiti.
- Branch Testing
  - Assicura che ogni possibile uscita da una condizione sia testata almeno una volta.

Test di equivalenza

Con questa tecnica siamo in grado di rendere al minimo il numero di test di case.
Consiste in due fasi:

1. Identificazione delle classi di equivalenza
2. Selezione degli input.

I possibili input sono suddivisi in classi di equivalenza e viene selezionato un caso di test per ogni classe. Si adotta con questo procedimento poichè il sistema, di solito, si comporta in modo simile per tutti i membri facenti parte di una stessa classe. Per testarlo quindi è necessario solo di prendere un membro ,che copra l'intera classe, per ogni classe di equivalenza.
I seguenti criteri vengono utilizzati per determinare le classi di equivalenza:

- Copertura: ogni possibile input appartiene a una delle classi di equivalenza.
- Disgiunzione: nessun input appartiene a più di una classe di equivalenza.
- Rappresentazione: se l'esecuzione mostra uno stato errato per un membro selezionato in una classe di equivalenza, è lecito pensare che lo stesso stato errato può essere rilevato selezionando un qualsiasi altro membro della stessa classe di equivalenza.

Una volta trovati le classi di equivalenza, si procede ad selezionare un valore valido che generi il caso normale e un valore non valido che generi la gestione delle eccezioni.
Vi sono due modi per usare equivalence testing:

1. Weak Equivalence Class Testing: prendiamo una variabile di ogni classe di equivalenza
2. Strong Equivalence Class Testing: basato sul prodotto cartesiane di un sottoinsiemi di partizioni in quale testiamo tutte le interazioni.

Test Case	a	b	c
WE1	a1	b1	c1
WE2	a2	b2	c2
WE3	a3	b3	c1
WE4	a1	b4	c2

Test Case	a	b	c
SE1	a1	b1	c1
SE2	a1	b1	c2
SE3	a1	b2	c1
SE4	a1	b2	c2
SE5	a1	b3	c1
SE6	a1	b3	c2
SE7	a1	b4	c1
SE8	a1	b4	c2
SE9	a2	b1	c1
SE10	a2	b1	c2
SE11	a2	b2	c1
SE12	a2	b2	c2
SE13	a2	b3	c1
SE14	a2	b3	c2
SE15	a2	b4	c1
SE16	a2	b4	c2
SE17	a3	b1	c1
SE18	a3	b1	c2
SE19	a3	b2	c1
SE20	a3	b2	c2
SE21	a3	b3	c1
SE22	a3	b3	c2
SE23	a3	b4	c1
SE24	a3	b4	c2

Se le condizioni di errore sono prioritarie, dovremmo estendere i test delle classi di equivalenza per includere classi non valide. Inoltre possiamo vedere dalle due foto che il numero di test case con il metodo WECT è sicuramente minore di quello di SECT.
WECT è appropriato quando i dati di input sono definiti in termini di intervalli e insiemi di valori discreti.
SECT presuppone che le variabili siano indipendenti.
Esempio da inserire

Boundary Testing

Un caso speciale di equivalence testing dove vengono considerati i valori al limite delle classi di equivalenza. Gli sviluppatori spesso trascurano casi speciali al limite della classi di equivalenza (ad esempio: 0, stringhe vuote, anno 200)

Il svantaggio di queste due tecniche è che non esplorano combinazioni di dati di input del test, in quanto un programma ha esito negativo perché una combinazione di determinati valori provoca l'errore errato.

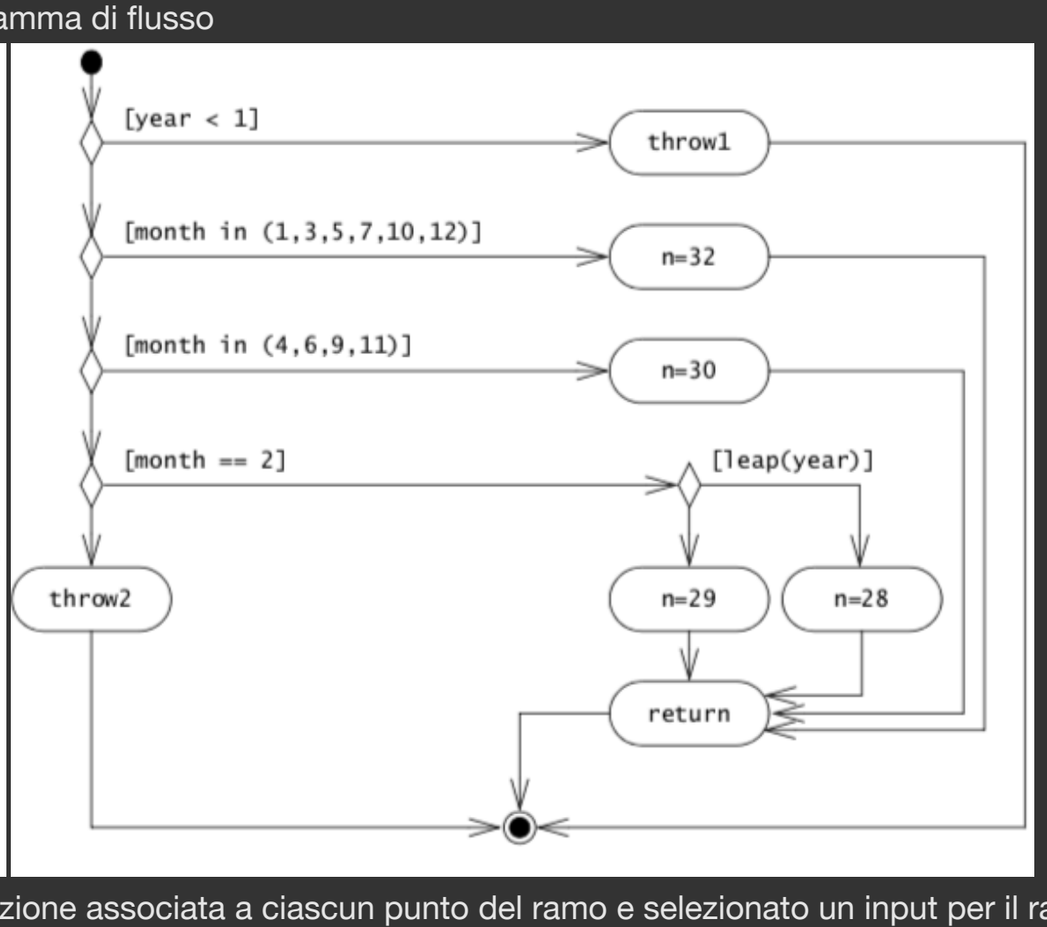
Path Testing

Metodo adottato con la whitebox dove identifica i guasti andando a vedere la struttura del codice sorgente. Lo scopo di questo test è che esercitando tutti i possibili percorsi attraversi il codice almeno una volta rivelando quindi dei fault che causeranno il failure. Come lo si dimostra dal nome whitebox, usare tale tecnica richiede la conoscenza del codice sorgente e delle strutture di dati. Per facilitarci alla costruzione dei casi di test, ci viene in aiuto "il diagramma di flusso". Tale diagramma è costituito da nodi che rappresentano blocchi eseguibili e bordi che rappresentano il flusso di controllo. Quindi ogni dichiarazione di decisione (es. while, if ecc) vengono mappati ai bordi mentre le istruzioni tra ogni decisione ai nodi.

Esempio: Mappatura da un codice sorgente a un diagramma di flusso

```
public class MonthOutOfBounds extends Exception {…};
public class YearOutOfBounds extends Exception {…};

class MyGregorianCalendar {
    public static boolean isLeapYear(int year) {
        boolean leap;
        if ((year%4) == 0){
            leap = true;
        } else {
            leap = false;
        }
        return leap;
    }
    public static int getNumDaysInMonth(int month, int year)
            throws MonthOutOfBounds, YearOutOfBounds {
        int numDays;
        if (year < 1) {
            throw new YearOutOfBounds(year);
        }
        if (month == 1 || month == 3 || month == 5 || month == 7 ||
                month == 10 || month == 12) {
            numDays = 32;
        } else if (month == 4 || month == 6 || month == 9 || month == 11) {
            numDays = 30;
        } else if (month == 2) {
            if (isLeapYear(year)) {
                numDays = 29;
            } else {
                numDays = 28;
            }
        } else {
            throw new MonthOutOfBounds(month);
        }
        return numDays;
    }
}
```



La verifica dei percorsi viene fatto esaminando la condizione associata a ciascun punto del ramo e selezionato un input per il ramo vero e un altro input per un ramo falso. Questa verifica completa del percorso ci garantisce la completezza del codice nel senso che ogni ramo di decisione viene passato almeno una volta.
Tuttavia non garantisce comunque al 100% la rivelazione dei guasti poiché il path testing tiene conto soltanto dei percorsi e non dei input particolari che possono innescare i fault come nel caso di equivalence testing (anno 2000 che è un anno bisestile).

Integration Testing

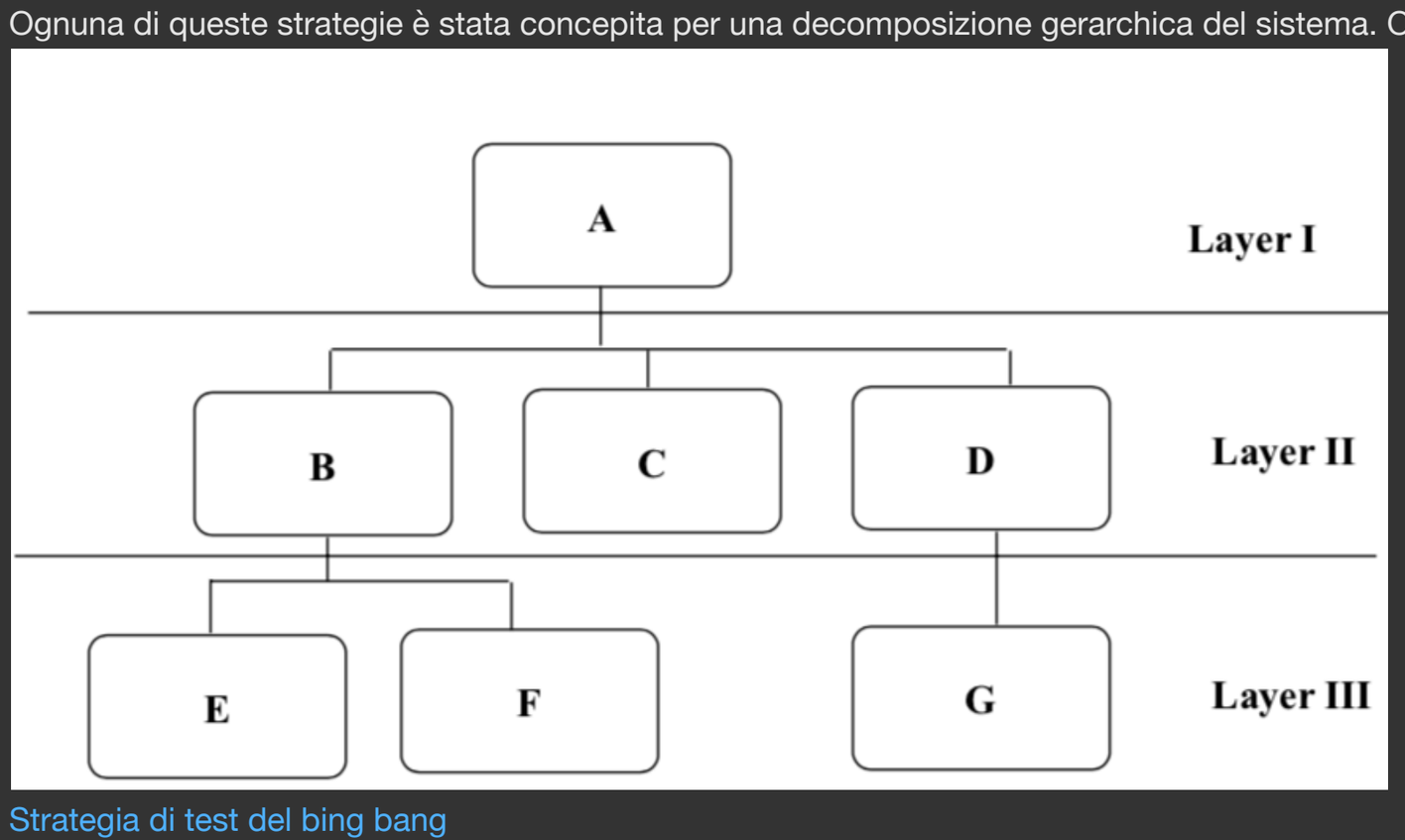
Una volta convalidati i singoli componenti testati nella fase Unit Testing, ovvero una volta stabiliti che nessuno di essi provochi guasti, si procede alla fase di Integrazion Testing. In tale fase, i componenti vengono integrati in sottosistemi più grandi andando a rilevare dei guasti che non sono stati rilevati nella fase Unit Testing a causa di un'implementazione scarsa di driver e stub. Questo perché nell'Unit testing i driver e i stub non simulavano completamente l'oggetto desiderato.
Con questa procedura, ci consente di testare parti sempre più complesse del sistema mentendo piccola la posizione dei potenziali guasti (ovvero il componente aggiunto più di recente è di solito quello che attiva i guasti rilevati più di recente).
Lo sviluppo di stub e driver di test per un test di integrazione richiede tempo. Per questo motivo, Extreme Programming prevede che i driver vengano scritti prima che i componenti vengano sviluppati.
Tuttavia l'ordine in cui i componenti vengono testati influisce molto lo sforzo richiesto dal test di integrazione quindi è consigliare di stilare un ordine correttamente in modo da ridurre le risorse necessarie per tale test di integrazione.

Strategie di test di integrazione orizzontale

Sono stati ideati diversi approcci per implementare una strategia di test di integrazione orizzontale:

- Test Big Bang
- Test Bottom-Up
- Test Top-Down
- Test Sandwich

Ognuna di queste strategie è stata concepita per una decomposizione gerarchica del sistema. Ogni componente appartiene ad una gerarchia di layer, ordinati in base all'associazione "Call".



Strategia di test del bing bang

Pressupone che vengano testati singolarmente tutti i componenti per poi essere testati insieme come un unico sistema.
il vantaggio è che non c'è bisogno di stub o di driver aggiuntivi. Tuttavia tali test sono costosi in quanto se un test rileva un errore, è molto difficile distinguere il fault tra i vari fault che si trovino all'interno di un componente. Inoltre è difficile individuare il componente specifico che lo ha causato il fault dato che tutti i componente sono esercitati.

Strategia di test bottom-up

Testa prima ogni singolo componente del livello inferiore, quindi li integra con i componenti del livello successivo. Questo viene ripetuto fino a quando i componenti di tutti i livelli sono combinati. I driver di test vengono utilizzare per simulare i componenti di livelli superiore che non sono ancora integrati. Non sono necessari i stub di test in tale fase.

Strategia di test top-down

Testa prima i componenti del layer superiore, quindi integra i componenti del livello inferiore. Quando tutti i componenti del nuovo ilivello sono stati testati insieme, viene selezionato il livello successivo. Ciò viene ripetuto fin a quando tutti i livelli non vengono combinati e coinvolti nel test. Gli stub di test vengono utilizzati per simulare i componenti degli strati inferiori che non sono ancora integrati. Non sono necessari i test driver.

Vantaggio&Svantaggio bottom up

il vantaggio del test bottom-up è che gli errori di interfaccia possono essere trovati più facilmente. Quando gli sviluppatori implementano i driver del livello successivo, hanno un idea chiara di come funziona il componente del livello corrispondente. Se il componente del livello superiore generi un fault, sarà molto facile trovarli.
Lo svantaggio dei test bottom-up è che verifica i sottosistemi più importanti per ultimi (i componenti dell'interfaccia utente). Di solito i guasti rilevati nel livello superiore possono portare a cambiamenti nella decomposizione del sistema o nelle interfacce del sottosistema dei livelli inferiori invalidando i test precedenti.

Vantaggio&Svantaggio top up

Il vantaggio del test top-down è che inizia con i componenti dell'interfaccia utente.
Lo svantaggio è che lo sviluppo di stub di test richiede tempo ed è soggetto a errori. E' richiesto un numero elevato di stub nel caso in cui il sistema non sia banale specialmente quando il livello più basso della decomposizione implementa molti metodi.

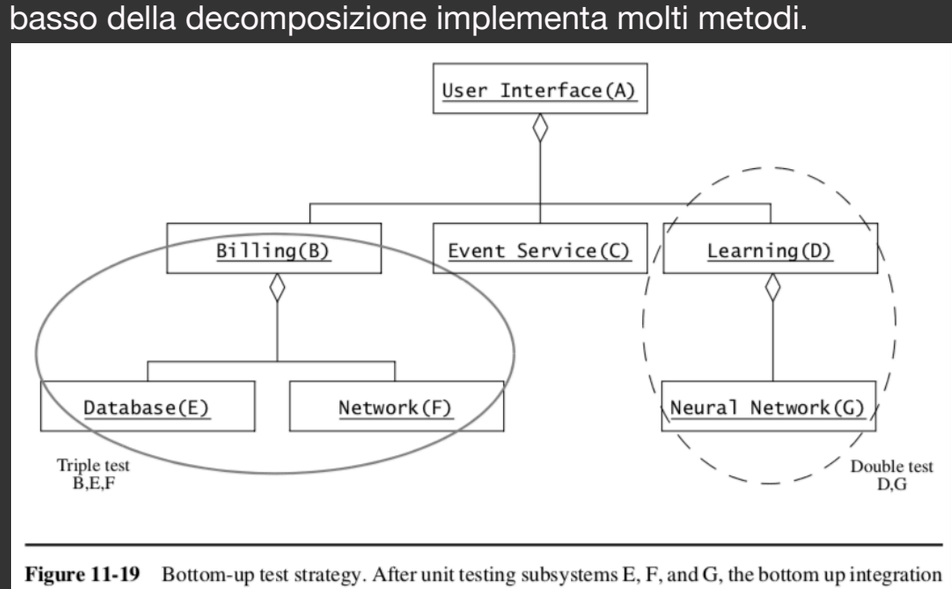


Figure 11-19 Bottom-up test strategy. After unit testing subsystems E, F, and G, the bottom up integration test proceeds with the triple test B-E-F and the double test D-G.

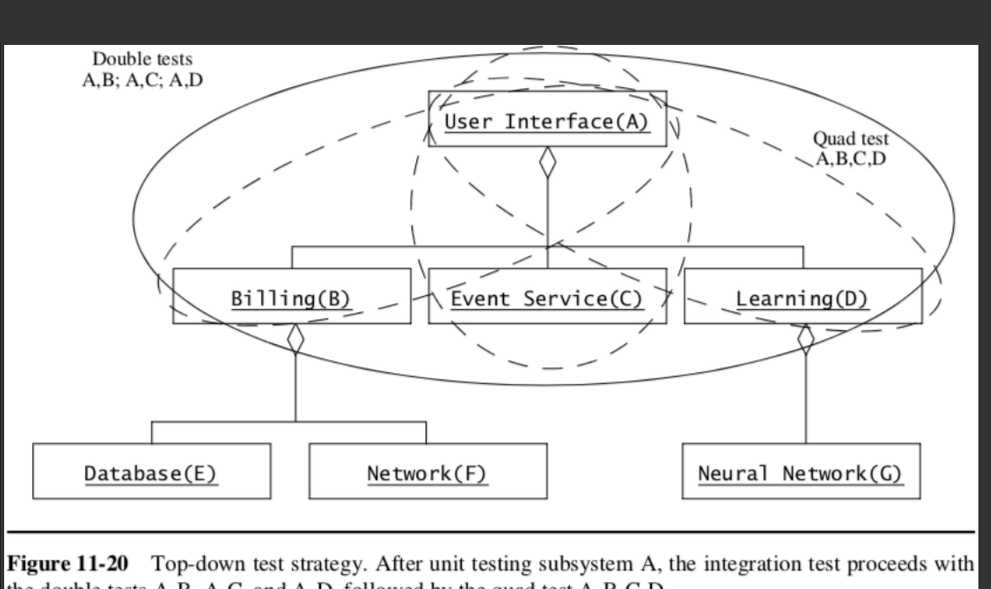


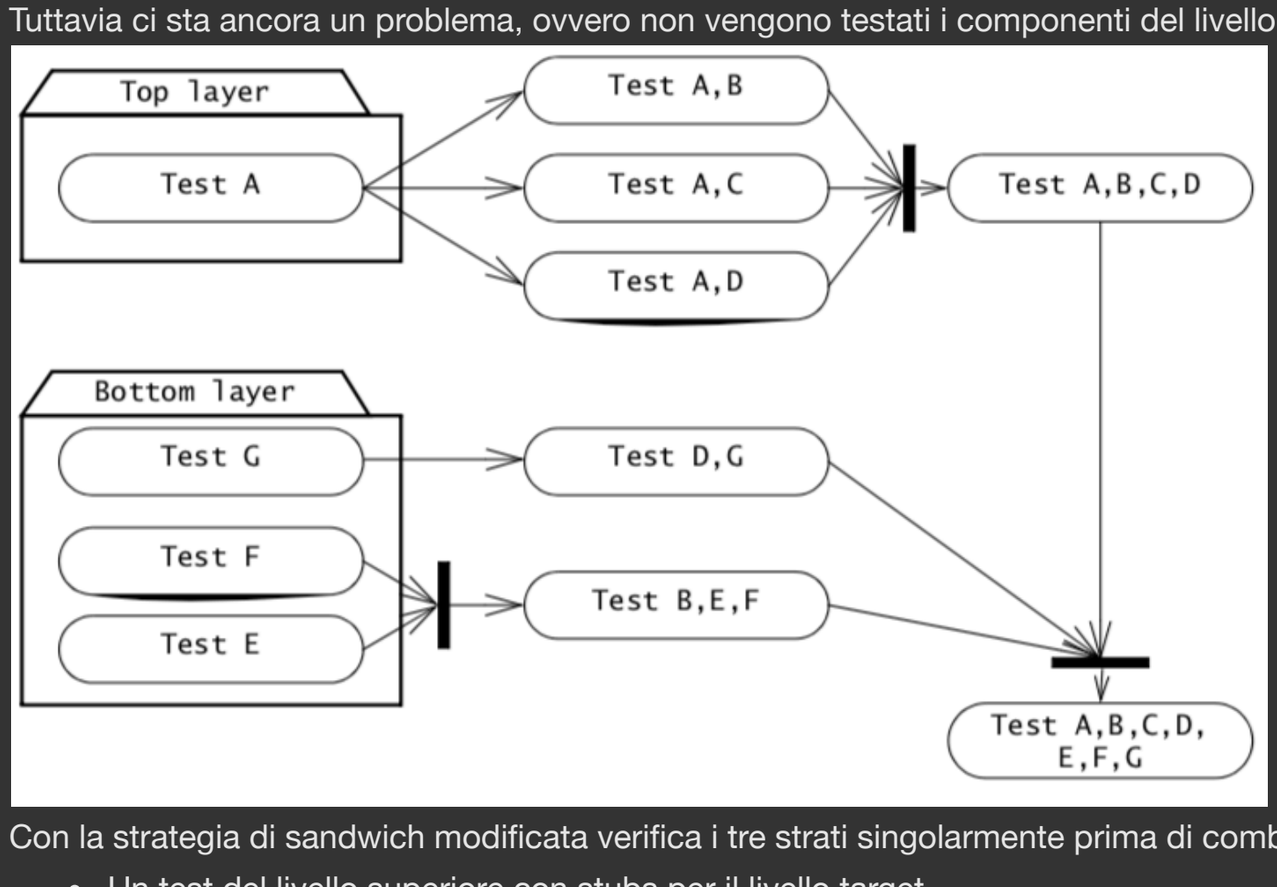
Figure 11-20 Top-down test strategy. After unit testing subsystem A, the integration test proceeds with the double tests A-B, A-C, and A-D, followed by the quad test A-B-C-D.

La strategia di test a sandwich

Combina le strategie top-down e bottom-up, cercando di sfruttare il meglio di entrambi.
Lo sviluppatore mappa la decomposizione del sistema in tre strati:

1. livello sopra il target (la fetta di pane superiore)
2. livello target (la succulenta carne)
3. livello sotto il targher (la fetta di pane inferiore)

In questo modo siamo in grado di eseguire due test contemporaneamente, bottom-up e top-down utilizzando il livello target come focus dell'attenzione. Il test di integrazione top-down viene eseguito testando il livello superiore in modo incrementale con i componenti del livello target e il test bottom-up viene utilizzato per testare il livello inferiore in modo incrementale con i componenti del livello target.
Tuttavia ci sta ancora un problema, ovvero non vengono testati i componenti del livello target.

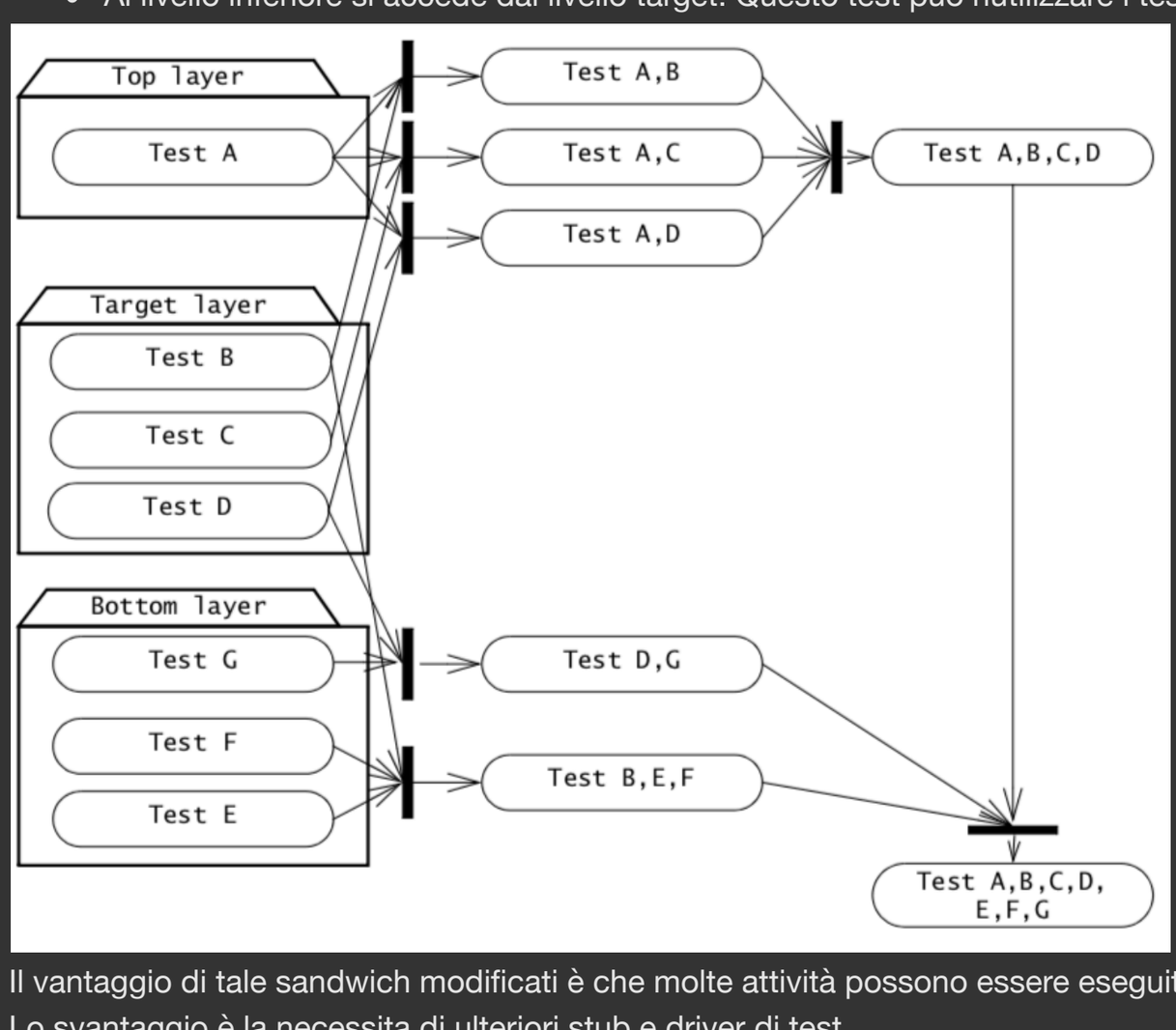


Con la strategia di sandwich modificata verifica i tre strati singolarmente prima di combinarli in test incrementali tra loro. I test dei singoli livelli sono costituiti da un gruppo di tre test:

- Un test del livello superiore con stubs per il livello target
- un test del livello target con driver e stubs che sostituiscono i livelli superiore e inferiore
- un test deli livello inferiore con i driver per il livello target.

I test a strati combinati consistono in due test:

- Il livello superiore accede al livello target. Questo test può riutilizzare i test del livello target dai test del singolo livello, sostituendo i driver con componenti del livello superiore.
- Al livello inferiore si accede dal livello target. Questo test può riutilizzare i test del livello target dai test del singolo livello, sostituendo i stubs con componenti del livello inferiore.



Il vantaggio di tale sandwich modificati è che molte attività possono essere eseguito in parallelo.
Lo svantaggio è la necessita di ulteriori stub e driver di test.
Nel complesso, i test sandwich modificati comportano un tempo di test complessivo più breve rispetto ai test top-down o bottom-up.

System Testing

Una volta integrati i componenti, i test di sistema garantiscono che l'intero sistema sia conforme ai requisiti funzionali e non funzionali.
Durante i test di sistema, vengono eseguite diverse attività:

- Test funzionali
- Test delle prestazioni
- Test pilota
- Test di accettazione
- Test di installazione

Category Partition

L'obiettivo è ridurre il numero (esponenziale) di combinazione considerando solo i casi più significativi.
E' una sistematizzazione del partizionamento di equivalenza e analisi del boundary.
Con il Category partition, adotteremo un approccio di circa 8 passi per passare dai requisiti ai test case.

1. Analizzare specifiche.
   - Il tester identifica ogni unita funzionale che può essere testata separatamente.
   - Esempio: findPrice—> recupero prezzo dal database.
2. Identificare categorie.
   - Per ogni unita testata, vengono individuati i parametri e per ogni parametro vengono creati delle categorie distinte.
   - Esempio: Categorie
     - codice: lunghezza, cifre più a sinistra, cifre rimanenti.
     - quantità: valore intero
     - peso: valore interno
     - database: contenuto
3. Partizione di categorie
   - Per ogni categoria, vengono identificati diversi casi rispetto ai quale testare le unità funzionali.
   - Esempio:
     - Code:
       - lunghezza: valido(8 lettere digitali), invalido (< or > 8)
       - cifre più a sinistra: 0,2,3,5, altri
       - cifre rimanenti: stringa valida, stringa non valida
     - Quantità:
       - valido, non valido
     - Peso:
       - valido, non valido
4. Identificare vincoli.
   - Vengono individuati i vincoli che esistono tra le scelte, ossie in che modo l'occorrenza di una scelta può influenzare l'esistenza di un'altra scelta
5. Valutare l'output del generatore:
   - Vengono generati dei test frames che consistono di combinazioni valide di scelte nelle categorie.
6. Generare gli script di test
   - Dai test frames vengono convertiti in test data.