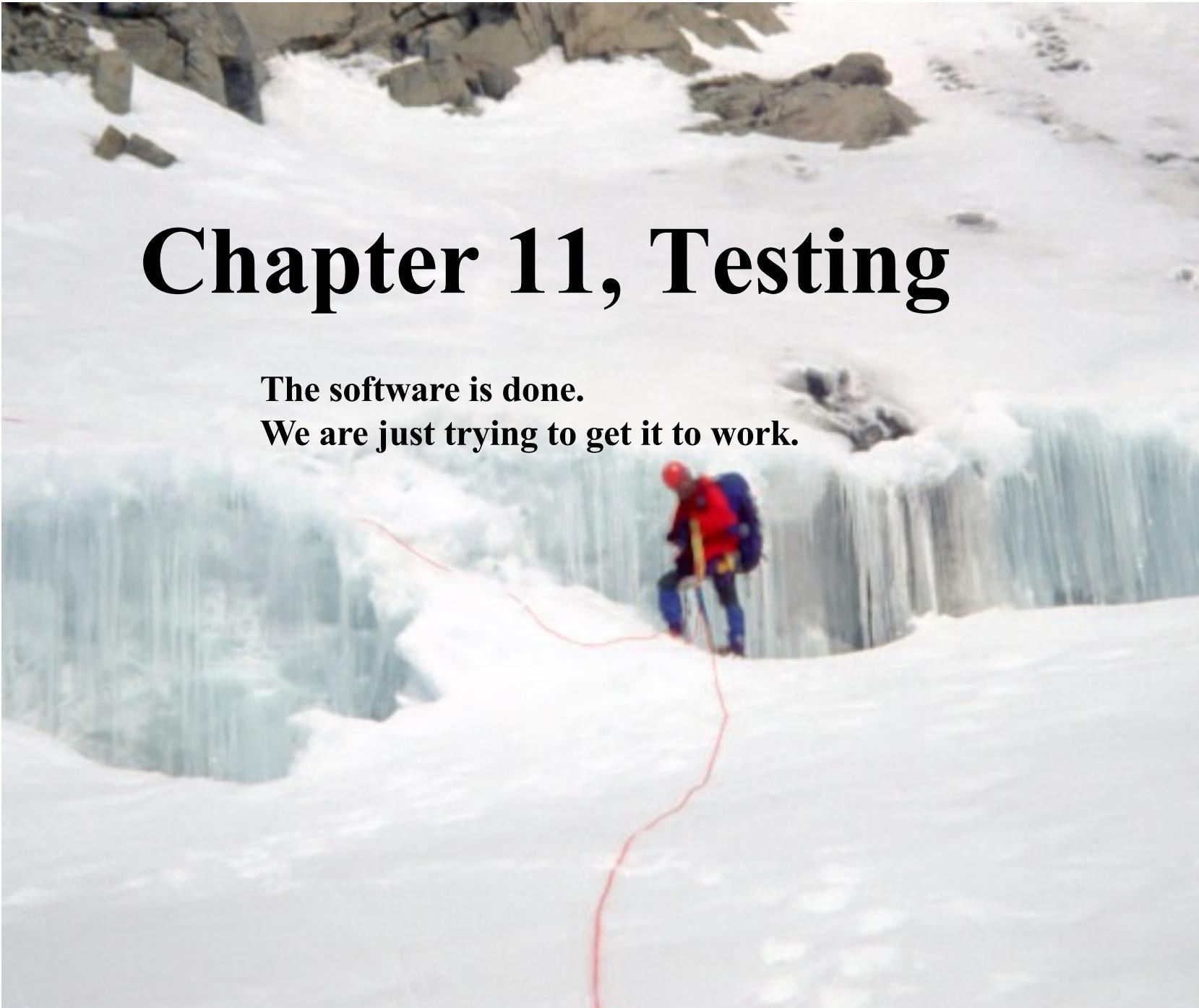


Chapter 11, Testing

**The software is done.
We are just trying to get it to work.**



Overview

- ◆ Il testing consiste nell'identificare **differenze** tra:
 - **Comportamento atteso**: specificato attraverso modelli o documenti di progettazione.
 - **Comportamento osservato**: rilevato nel sistema implementato.
- ◆ **Unit** testing
 - ◆ Trovare differenze tra **specifiche del componente** (es. **requisiti** o **design dettagliato**) e il **comportamento effettivo del singolo componente**.
- ◆ **Integration** testing
 - ◆ Trovare differenze tra **system design model** e un **sottoinsieme integrato di sottosistemi**
- ◆ **Functional** testing
 - ◆ Trovare differenze tra i **requisiti funzionali** definiti (ad esempio tramite **use case model**) e il **comportamento effettivo del sistema**
- ◆ **Performance** testing
 - ◆ Trovare differenze tra **requisiti non funzionali** (es. **tempi di risposta, scalabilità**) e le **performance del sistema reale**

Overview

- ◆ The Goal: **progettare test** per provare il sistema e **rivelare problemi**
 - ◆ **Massimizzare il numero di errori scoperti che consentirà agli sviluppatori di correggerli**
- ◆ Questa attività va in contrasto con le altre attività svolte prima: analisi, design, implementazione sono attività “costruttive”. Il testing invece tenta di “**rompere**” il sistema
- ◆ Il testing dovrebbe essere realizzato da sviluppatori che non sono stati coinvolti nelle attività di costruzione del sistema

Outline

- ◆ Terminologia
- ◆ Tecniche per gestire i difetti e gli errori
- ◆ Le attività di testing
- ◆ Component/Unit testing
- ◆ Integration testing
- ◆ System testing
- ◆ Gestione del testing

Quality of today's software....

- ❖ I prodotti software messi sul mercato non sono “error free”.

A Joke

- ❖ How cars would work if they followed a development history similar to that of computers
- ❖ At the COMDEX computer exposition, Bill Gates
 - ◆ **“If General Motors had kept up with technology like the computer industry has, we would all be driving \$25 cars that got 1,000 miles to the gallon.”**

General Motors' Reply

- ❖ “If GM had developed technology like Microsoft, we would all be driving cars with the following characteristics:
 - ◆ **For no reason whatsoever your car would crash twice a day.**
 - ◆ **Every time they repainted the lines on the road, you would have to buy a new car.**
 - ◆ **Macintosh would make a car that was powered by the sun, reliable, five times as fast, and twice as easy to drive, but would run on only five percent of the roads.**

General Motors' Reply

- ◆ New seats would force everyone to have the same size hips.
- ◆ The airbag system would say “Are you sure?” before going off.
- ◆ Occasionally, for no reason whatsoever, your car would lock you out and refuse to let you in until you simultaneously lifted the door handle, turned the key, and grabbed hold of the radio antenna.”



Is quality a real problem with IT projects?

- ❖ Most people simply accept poor quality from many IT products.
 - ◆ **So what if your computer crashes a couple of times a month? Just make sure you back up your data.**
 - ◆ **So what if you cannot log in to the corporate intranet or the Internet right now? Just try a little later when it is less busy.**
 - ◆ **So what if the latest version of your word-processing software was shipped with several known bugs? You like the software's new features, and all new software has bugs.**

Is quality a real problem with IT projects?

- ❖ Yes, it is! IT is not just a luxury available in some homes, schools, or offices.
- ❖ Many aspects of our daily lives depend on high-quality IT products.
 - ◆ **Food is produced and distributed with the aid of computers;**
- ❖ Many IT projects develop mission-critical systems that are used in life-and-death situations
 - ◆ **navigation systems on aircraft, computer components built into medical equipment...**

Is quality a real problem with IT projects?

- ❖ Financial institutions and their customers also rely on high-quality information systems.
 - ◆ **Customers get very upset when systems provide inaccurate financial data or reveal information to unauthorized users that could lead to identity theft.**
- ❖ When one of these systems does not function correctly, it is much more than a slight inconvenience!

WHAT WENT WRONG?

- ❖ Nel 1981, una piccola differenza di sincronizzazione causata da una modifica a un programma informatico creò una probabilità di 1 su 67 che i cinque computer di bordo dello Space Shuttle non si sincronizzassero. L'errore portò all'annullamento del lancio.
- ❖ Nel 1986, due pazienti ospedalieri morirono dopo aver ricevuto dosi letali di radiazioni da una macchina Therac 25. Un problema software causò che i dati di calibrazione furono ignorata da parte della macchina....

WHAT WENT WRONG ?

- ❖ Uno dei più grandi errori software nella storia bancaria:

Chemical Bank ha erroneamente detratto circa 15 milioni di dollari da più di 100.000 conti correnti.

Il problema derivava da **una singola riga di codice** in un programma aggiornato, che ha fatto sì che la banca elaborasse ogni prelievo e trasferimento ai suoi sportelli automatici (ATM) **due volte**. Ad esempio, una persona che prelevava 100 dollari da un ATM si vedeva detrarre 200 dollari dal conto, anche se la ricevuta indicava solo un prelievo di 100 dollari. L'errore ha coinvolto 150.000 transazioni.

Software quality management

- ❖ Si occupa di garantire che il livello di qualità richiesto venga raggiunto in un prodotto software.
- ❖ Implica la definizione di **standard e procedure di qualità appropriati** e **l'assicurazione che vengano seguiti**.
- ❖ Dovrebbe mirare a sviluppare una "**cultura della qualità**" in cui la qualità sia vista come una responsabilità di tutti.

Perché? Che cosa? Quando?

- ❖ GOAL: software con **zero difetti** ...
- ❖ MA **impossibile** da ottenere e garantire
- ❖ Necessaria una attenta e continua **VERIFICA e CONVALIDA (V&V)**
- ❖ Tutto deve essere verificato: documenti di specifica, di progetto, dati di collaudo,programmi
- ❖ Si fa lungo tutto il processo di sviluppo, **NON** solo alla fine!

Verification & Validation (V&V)

- ❖ **Verifica** (*verification*):
 - insieme delle attività volte a stabilire se il sw costruito **soddisfa le specifiche** (non solo funzionali)
 - did we build the program right?
 - ❖ si assume che le specifiche esprimano in modo esauriente e corretto i *desiderata del committente*
 - ❖
 - ❖ **Convalida** (*validation*):
 - stabilire che il sistema soddisfa le **esigenze vere** dell’utente
 - did we build the right program ?
 - ❖ Può essere svolta sulla specifica (meglio!) e / o sul sistema finale

Prova di correttezza e testing

- ❖ Prova di correttezza:
 - ◆ **–argomentare sistematicamente (in modo formale o informale) che il programma funziona correttamente per *tutti i possibili dati di ingresso***
- ❖ Testing:
 - ◆ particolare tipo di attività sperimentale fatta mediante **esecuzione** del programma, selezionando **alcuni dati di ingresso** e valutando risultati
 - ◆ dà un riscontro **parziale**: *programma provato solo per quei dati*
 - ◆ Tecnica **dinamica** rispetto alle *verifiche statiche fatte dal compilatore*

Testing

- ❖ Program testing can be used to show the *presence of bugs, but never to show their absence*. (*Dijkstra 1972*)
- ❖ Quindi obiettivo del testing è di **trovare "controesempi"**
- ❖ si cerca di:
*trovare dati di test che massimizzino la probabilità
di scoprire errori durante l'esecuzione*

Terminology

- ◆ **Affidabilità:** La misura di successo con cui il comportamento osservato di un sistema è **conforme** ad una certa specifica del relativo comportamento.
- ◆ **Fallimento (failure):** Qualsiasi **deviazione** del comportamento osservato dal comportamento specificato.
- ◆ **Stato di Errore:** Il sistema è in uno stato tale che **ogni ulteriore elaborazione** da parte del sistema porta ad un fallimento.
- ◆ **Difetto (Bug/fault):** La **causa** meccanica o algoritmica di un failure.

Fault e Failure

```
int raddoppia(x)
{
    int y;
    y = x*x;
    return (y);
}
```

Per il valore di ingresso $x = 3$ si ha il valore di uscita $y = 9$. La causa di questa *failure* è il *fault* di linea 2, in cui anziché l'operatore + è usato l'operatore *

NB: Se il valore di ingresso è $x = 2$, il valore di uscita è $y = 4$ (*nessuna failure*)

Fault, Failure e Difetto

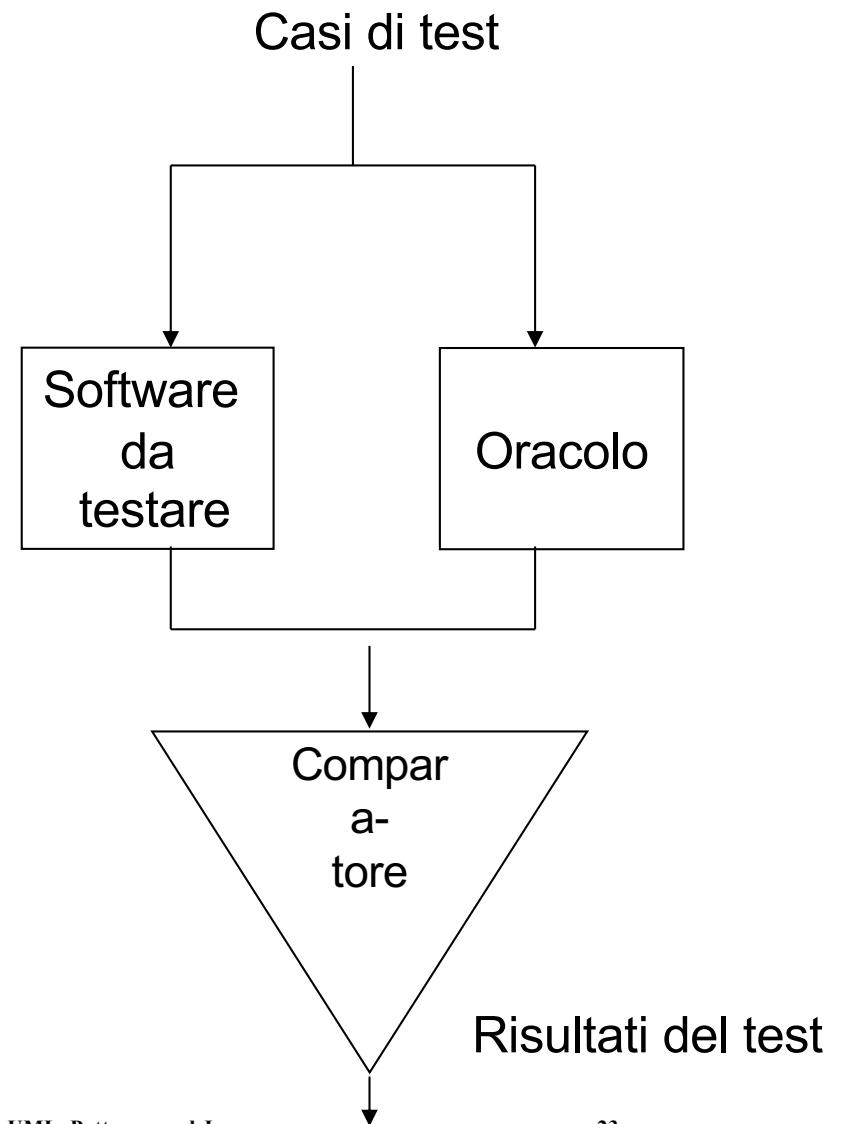
- ◆ **Non tutti i fault generano failure**
- ◆ **Una failure può essere generata da più fault**
- ◆ **Un fault può generare diverse failure**
- ◆ ***Defect*** (difetto) quando non è importante distinguere fra fault e failure si può usare il termine defect per riferirsi sia alla causa (fault) che all'effetto (failure)

Test e Casi di test

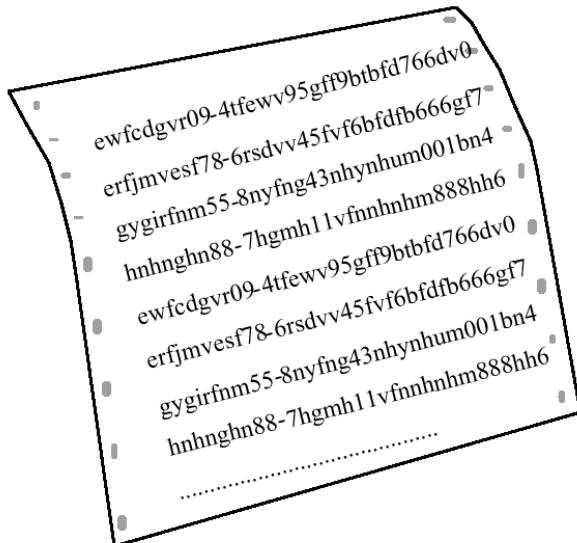
- ◆ Un programma è esercitato da un **caso di test** (insieme di dati di input e...risultato atteso...)
- ◆ Un **test** (o **test suite**) è formato da un **insieme di casi di test**
- ◆ **L'esecuzione del test** consiste nell'esecuzione del programma per tutti i casi di test della test suite
- ◆ Un test ha **successo** se **rileva** uno o più malfunzionamenti (failure) del programma

L'oracolo ...

- ◆ Condizione necessaria per effettuare un test:
 - ◆ **conoscere il comportamento atteso per poterlo confrontare con quello osservato**
- ◆ L'oracolo conosce il comportamento atteso per ogni caso di prova
- ◆ Oracolo umano
 - ◆ **si basa sulle specifiche o sul giudizio**
- ◆ Oracolo automatico
 - ◆ **generato dalle specifiche (formali)**
 - ◆ **stesso software ma sviluppato da altri**
 - ◆ **versione precedente (test di regressione)**



Un buon oracolo



- Il test di applicazioni grandi e complesse può richiedere milioni di casi di test
- La dimensione dello spazio di uscita può eccedere le capacità umane
- L'occhio umano è lento e poco affidabile anche per uscite di piccole dimensioni

ORACOLI AUTOMATICI SONO ESSENZIALI !

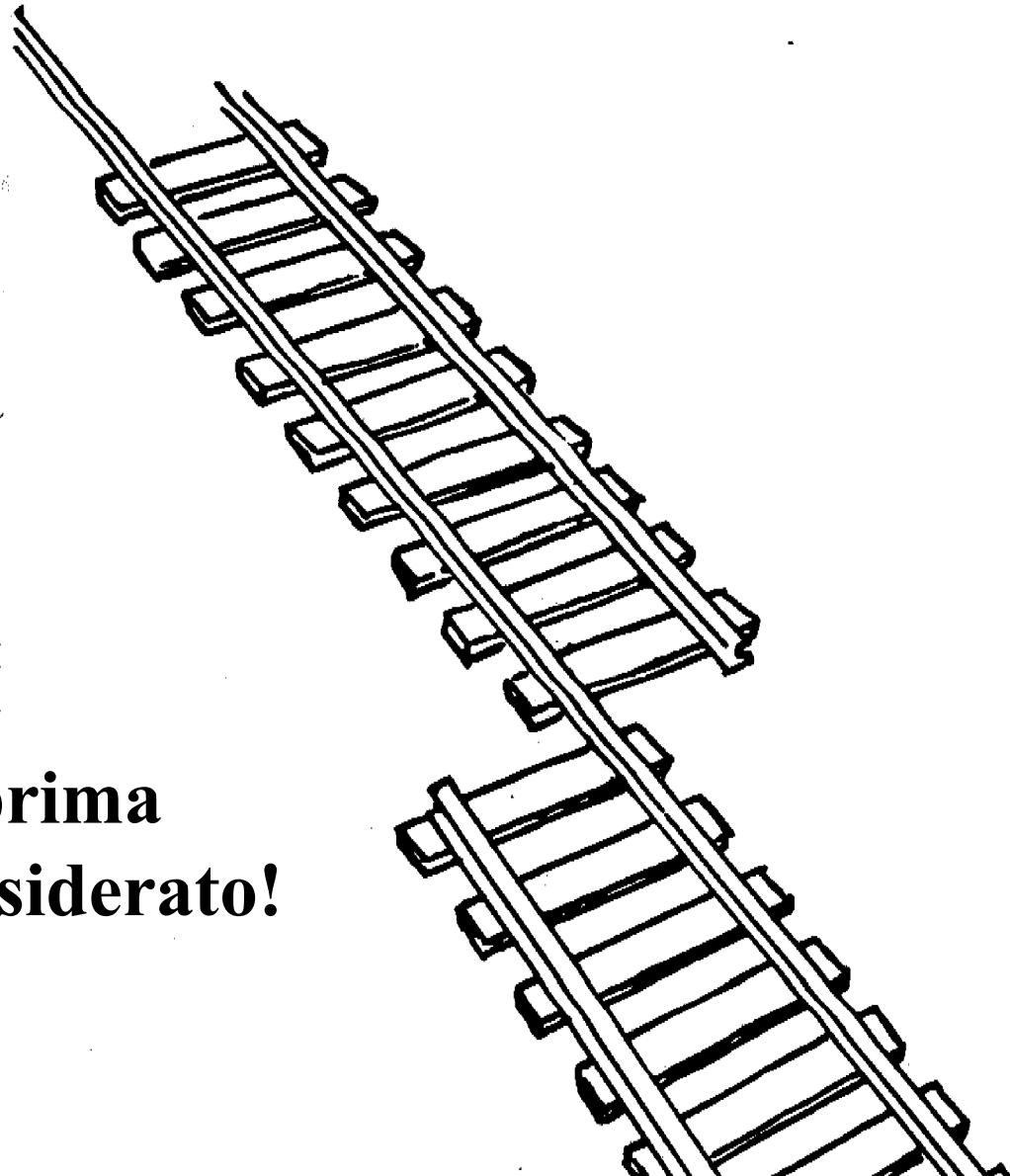
Cos'è?

Un failure?

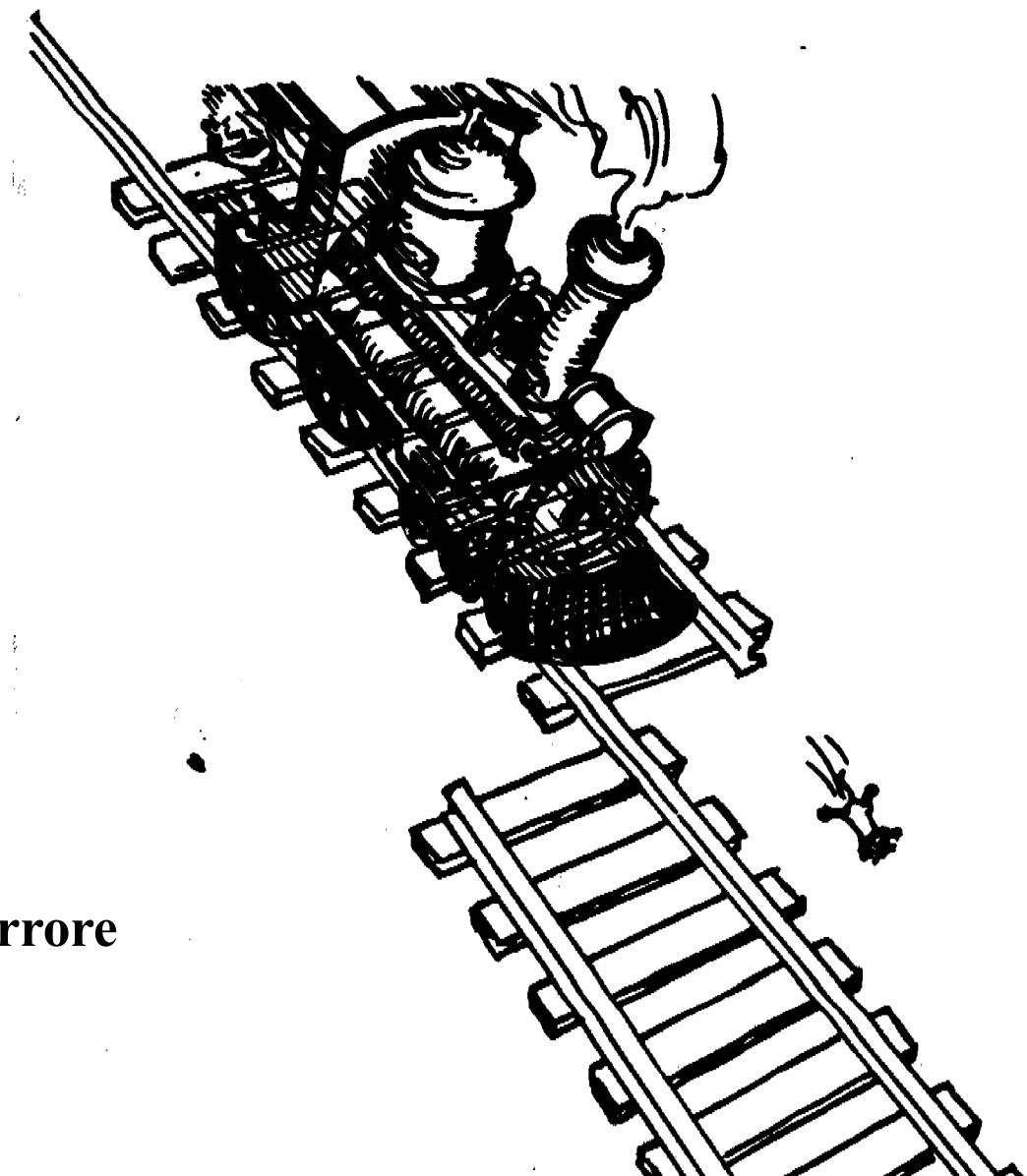
Un errore?

Un fault?

**Bisogna specificare prima
il comportamento desiderato!**



Erroneous State (“Error”)

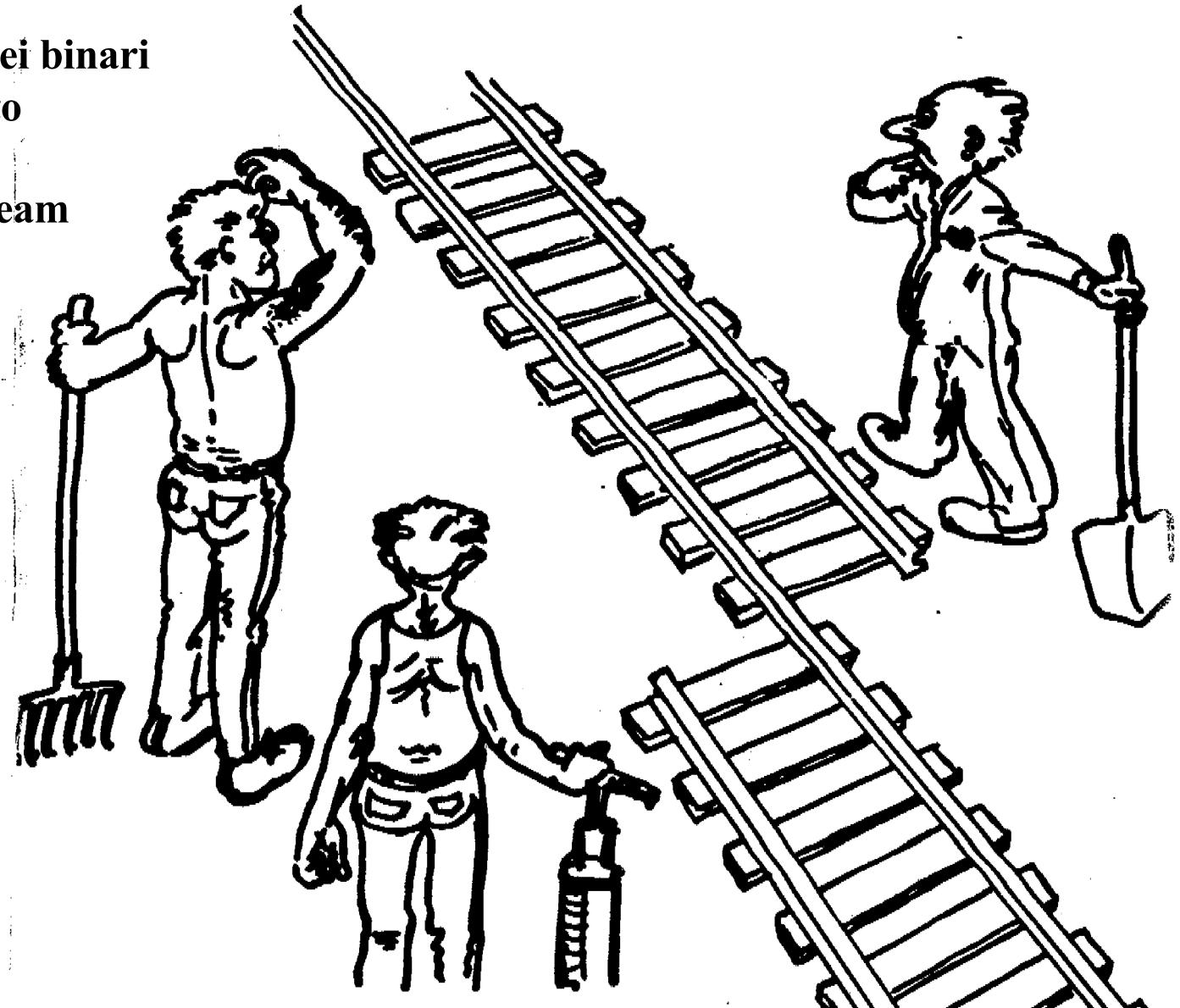


**Lo stato corrente mostra un errore
Ma non un fallimento!**

Algorithmic Fault

**Il non allineamento dei binari
può essere un risultato
della cattiva
comunicazione tra i team**

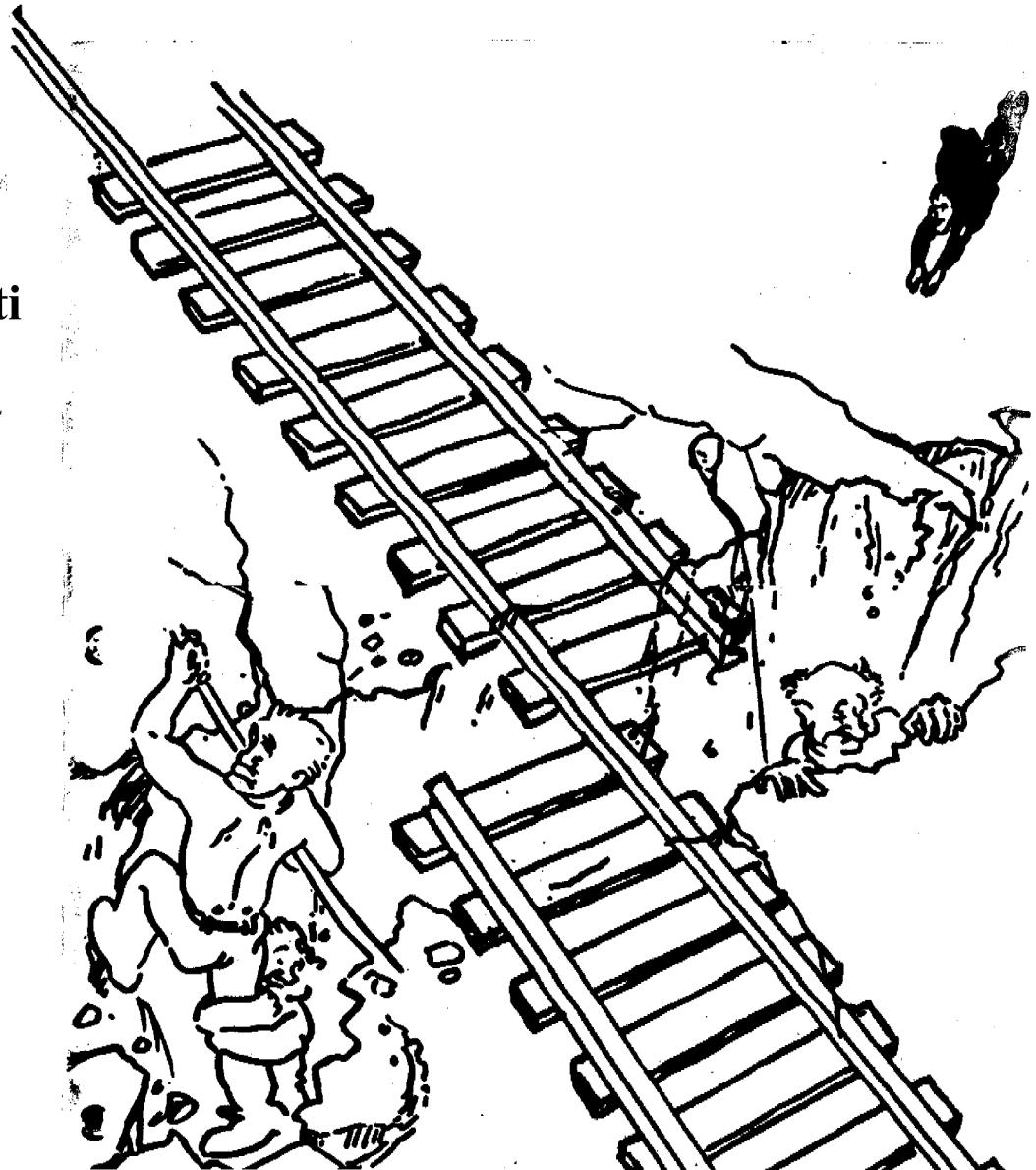
**Oppure
l'implementazione
sbagliata
della specifica
da parte di un team**



Mechanical Fault

Anche se i binari sono implementati in accordo alle specifiche nel RAD potrebbero risultare non allineate nella messa in opera, per esempio, a causa di un terremoto

Un fallimento della virtual machine di un sistema software è un altro esempio di fallimento meccanico: anche se lo sviluppatore ha implementato correttamente, cioè mappato correttamente l'object model nel codice, il comportamento osservato può deviare ancora da quello atteso

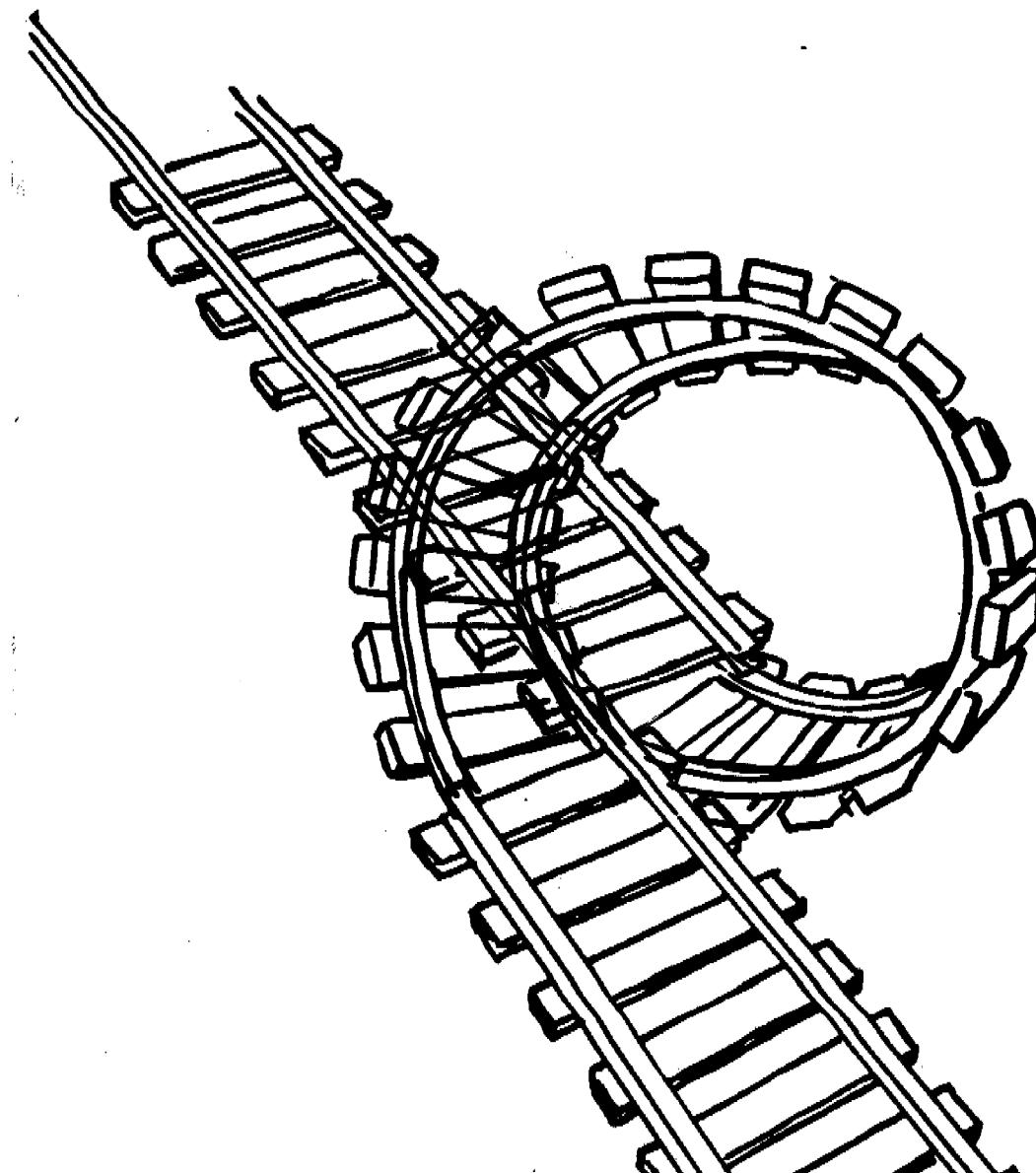


Esempi: Fault e Errori

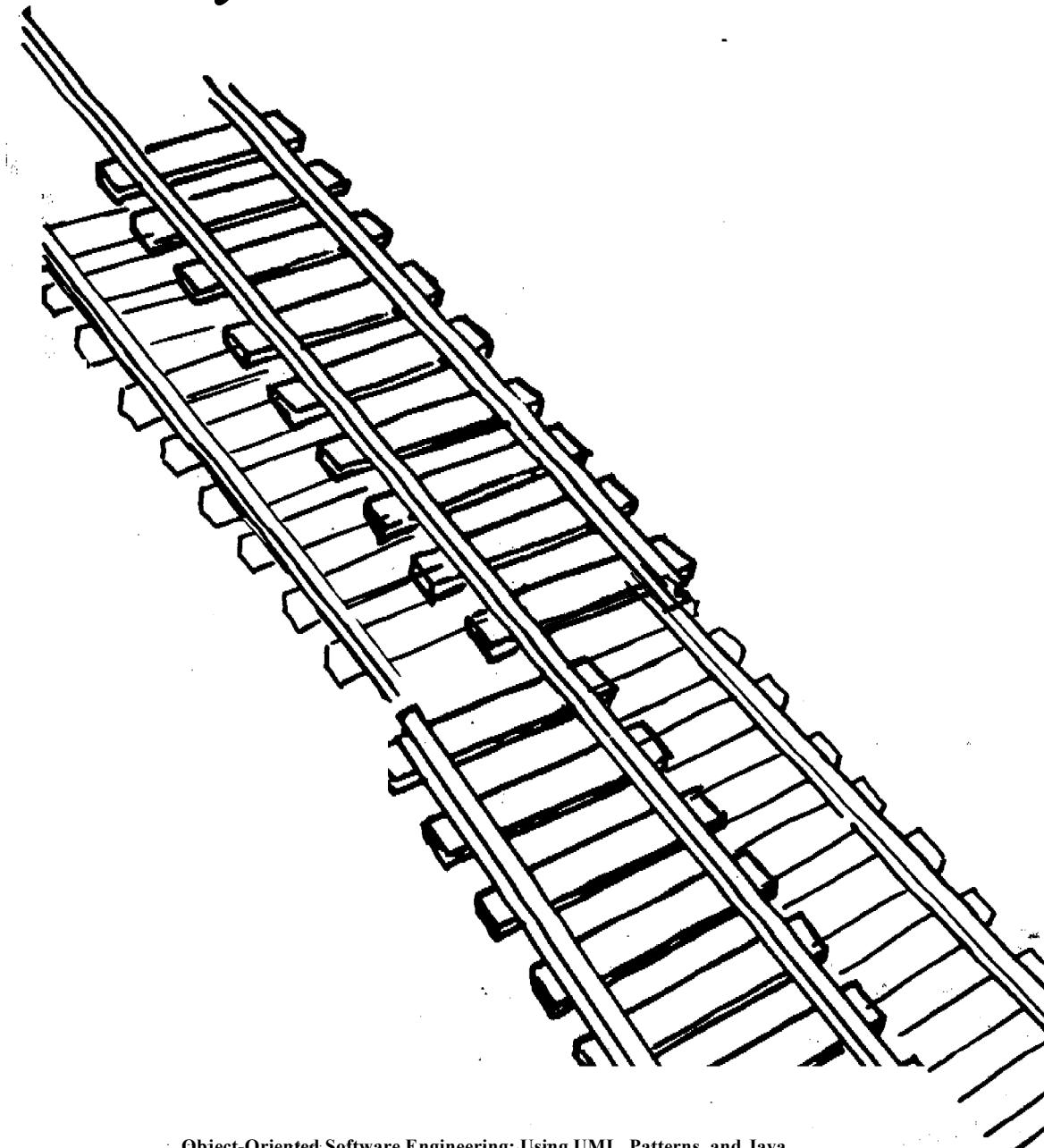
- ◆ Fault nella specifica delle interfacce
 - ◆ **Disallineamento tra quello di cui il client ha bisogno e cosa gli viene offerto**
 - ◆ **Disallineamento fra la specifica e l'esecuzione**
- ◆ Algorithmic Faults
 - ◆ **Inizializzazione mancante**
 - ◆ **Errori di ramificazione (uscita dal loop troppo presto, troppo in ritardo)**
 - ◆ **Prova mancante per zero**
- ◆ Mechanical Faults (difficili da trovare)
 - ◆ **La documentazione non è aderente alle condizioni reali e alle procedure operative**
- ◆ Errori
 - ◆ **Errori che scaturiscono da sovraccarico**
 - ◆ **Errori limite e di capacità**
 - ◆ **Errori legati al throughput o alla performance**

Come trattare gli errori e i difetti?

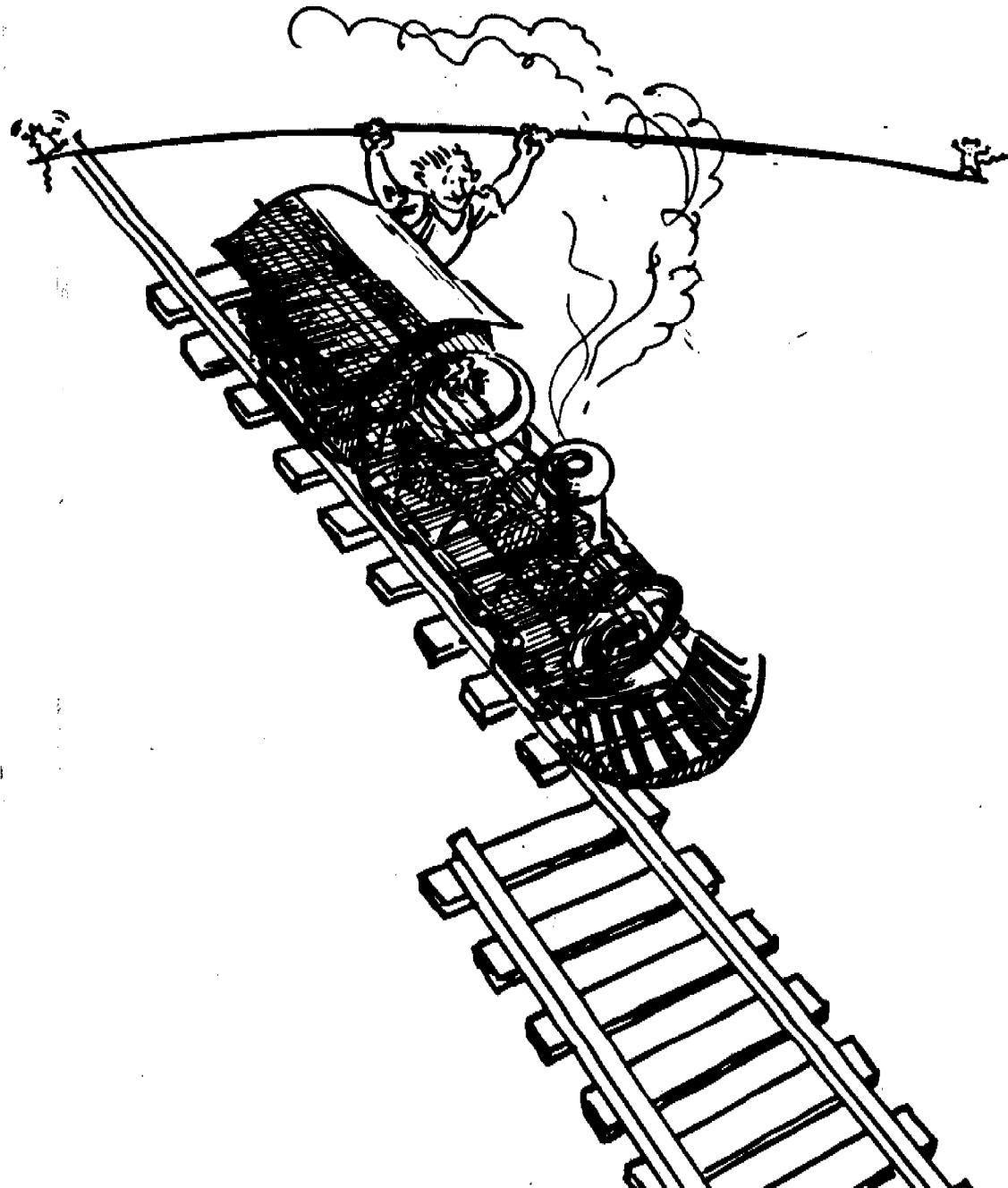
Verification?



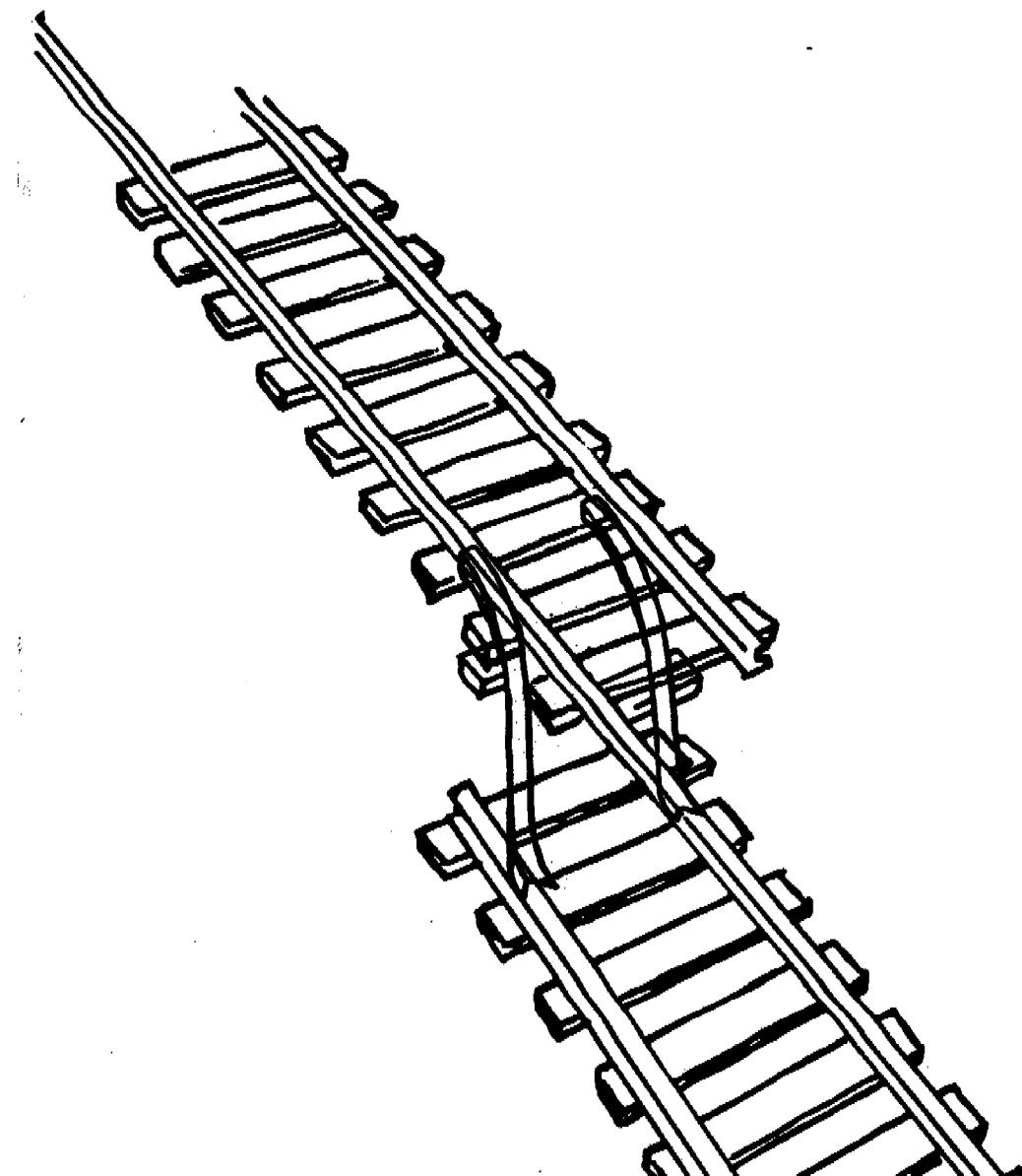
Modular Redundancy?



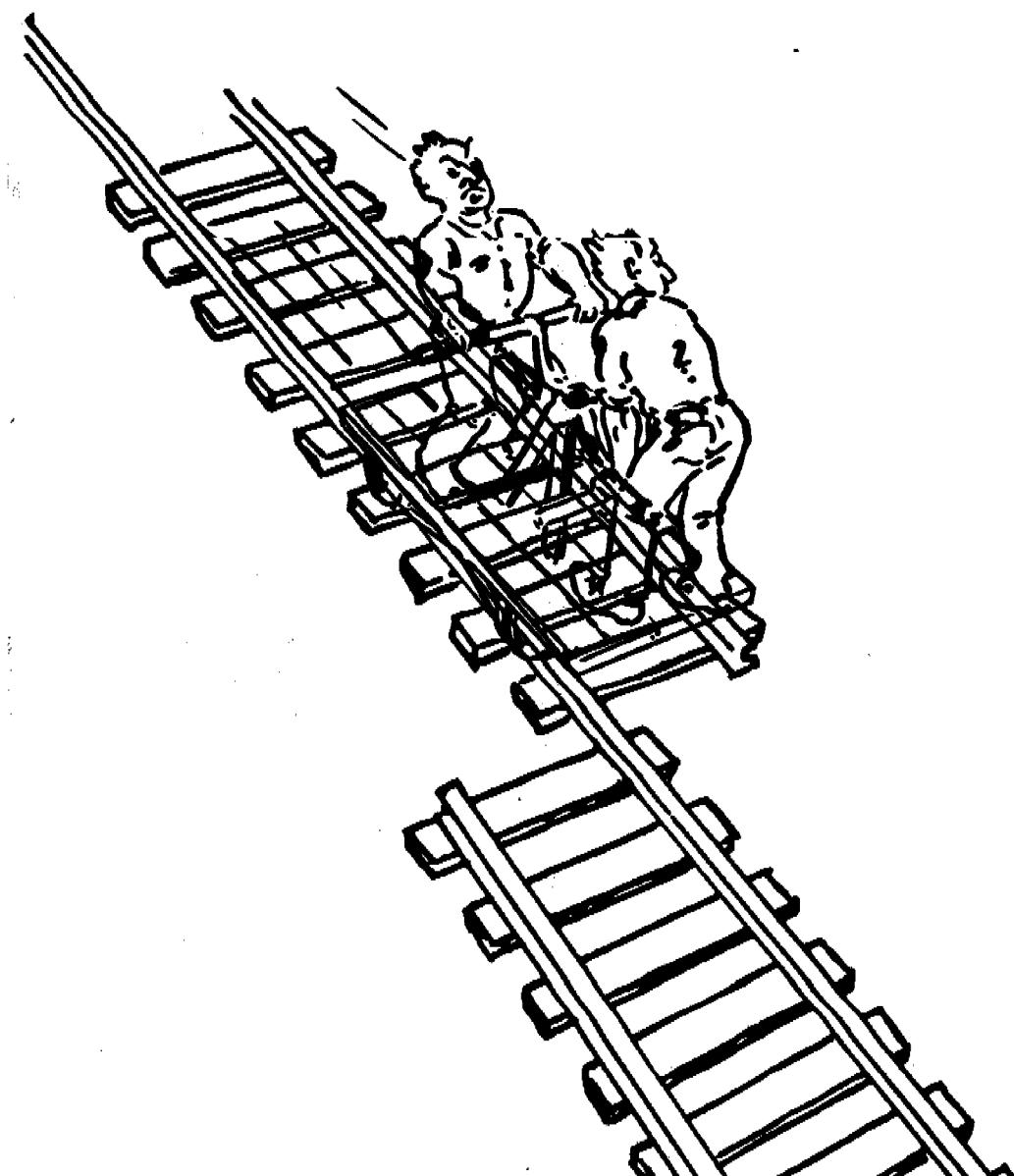
Declaring the Bug as a Feature?



Patching?



Testing?



Dealing with Defects

- ◆ **Verification:**

- ◆ Presuppone un ambiente ipotetico che non corrisponde all'ambiente reale.
La dimostrazione potrebbe essere difettosa (omette vincoli importanti; semplicemente errata)

- ◆ **Modular redundancy:**

- ◆ Expensive

- ◆ **Declaring a bug to be a “feature”**

- ◆ Bad practice

- ◆ **Patching**

- ◆ Slows down performance

- ◆ **Testing (this lecture)**

- ◆ Testing is never good enough

Tecniche per aumentare l'Affidabilità di un sistema software

- ◆ **Fault Avoidance.** Tecniche che tentano di prevenire l'inserimento di difetti nel sistema prima che sia realizzato
 - ◆ Includono metodologie di sviluppo, gestione delle configurazioni, e verifica
 - ◆ Use good programming methodology to reduce complexity
 - ◆ Use version control to prevent inconsistent system
 - ◆ Apply verification to prevent algorithmic bugs
- ◆ **Fault detection.** Tecniche come **debugging** e **testing**, sono rispettivamente esperimenti non controllati e controllati usati durante il processo di sviluppo per identificare stati di errore e trovare il difetto alla base. Includono anche **review**
 - ◆ Testing: Create failures in a planned way
 - ◆ Debugging: Start with an unplanned failures
 - ◆ Review: revisione manuale
- ◆ **Fault tolerance.** Assumono che un sistema possa essere realizzato con bug e che i fallimenti del sistema possano essere gestiti effettuando il recovery a run-time
 - ◆ Data base systems (atomic transactions)
 - ◆ Modular redundancy

Tecniche per aumentare l'Affidabilità di un sistema software

- ◆ Le tecniche **Fault detection**, includono anche *review*
- ◆ **Review:** ispezione manuale di alcuni o tutti gli aspetti del sistema senza eseguire realmente il sistema (molto efficace: 85% dei fault rilevati avvengono grazie a review)
- ◆ Due tipi di review: **walkthrough** e **inspection**.
 - ◆ **Walkthrough.** Lo sviluppatore presenta informalmente le API, il codice, la documentazione associata delle componenti al **team di review**. Il team commenta sul mapping modelli-codice usando RAD
 - ◆ **Inspection.** Simile al walkthrough, ma la presentazione delle unità è **formale da parte del team di review**.
 - ◆ Lo sviluppatore non può presentare gli artefatti. Questo è fatto dal team di review che è responsabile di controllare
 - le **interfacce** e il codice rispetto ai **requisiti**
 - I **commenti** rispetto al **codice**
 - **l'efficienza degli algoritmi** rispetto alle **richieste non funzionali**
 - ◆ Lo sviluppatore interviene solo se si richiedono chiarimenti

Tecniche per aumentare l'Affidabilità di un sistema software

1. **Debugging:** è il processo di individuazione e correzione dei bug in un sistema dopo che è stato rilevato un errore (failure). Parte dal presupposto che un **failure non pianificato** evidenzia un problema sottostante nel codice o nella logica. (è reattivo)
2. Il processo tipico include:
 1. Identificazione dello stato di errore: analisi di dove il sistema devia dal comportamento atteso.
 2. Tracciamento delle cause: attraversare gli stati corretti per risalire al punto esatto in cui si verifica il bug.
- Debugging sia per correttezza funzionale sia per performance
- ♦ **Testing.** Tecnica che tenta di **creare** stati di fallimento o errore in modo **proattivo e pianificato**.
- ♦ Questa definizione implica che per **successo** del testing si indica la situazione in cui siamo stati in grado di generare un **failure** e quindi identificare **fault**.
 - ◆ Un buon test contiene test case che creano failure e fanno identificare fault
 - ◆ I test case dovrebbero includere un range ampio di valori di input, incluso input invalidi, e condizioni limite (boundary)
 - ◆ questo approccio richiede molto tempo anche per il testing di sistemi di piccole dimensioni

Some Observations

- ◆ It is impossible to completely test any nontrivial module or any system
 - ◆ **Theoretical limitations: Halting problem**
 - ◆ Il settore del testing è tormentato da problemi indecidibili
 - ◆ un problema è detto *indecidibile* (irrisolubile) se è possibile dimostrare che non esistono algoritmi che lo risolvono ...
 - ◆ es. stabilire se l'esecuzione di un programma termina a fronte di un input arbitrario è un problema indecidibile
 - ◆ **Practical limitations: Prohibitive in time and cost**
- ◆ Testing can only show the presence of bugs, not their absence (Dijkstra)
- ◆ Testing should be integrated with other verification activities, e.g., inspections
- ◆ Pertanto, il testing si concentra nel **ridurre il rischio di errori critici**, piuttosto che eliminarli completamente.

Testing takes creativity

- ◆ Testing often viewed as dirty work.
- ◆ To develop an effective test, one must have:
 - ◆ **Detailed understanding of the system**
 - ◆ **Knowledge of the testing techniques**
 - ◆ **Skill to apply these techniques in an effective and efficient manner**
- ◆ Testing is done best by independent testers
 - ◆ **We often develop a certain mental attitude that the program should in a certain way when in fact it does not.**
- ◆ Programmer often stick to the data set that makes the program work
 - ◆ **"Don't mess up my code!"**
- ◆ A program often does not work when tried by somebody else.
 - ◆ **Don't let this be the end-user.**

Testing esaustivo

- ◆ Testing esaustivo (*esecuzione per tutti i possibili ingressi*) dimostra la correttezza
 - ◆ Es. se programma calcola un valore in base a un valore di ingresso nel range 1..10, testing esaustivo consiste nel provare tutti i valori: per le 10 esecuzioni diverse si verifica se il risultato è quello atteso
- ◆ ... impossibile da realizzare in generale:
 - ◆ Es. se programma legge 3 input interi nel range 1..10.000 e calcola un valore, testing esaustivo richiede 10^{12} esecuzioni ($10^4 \times 10^4 \times 10^4$)!

1 test / millisecondo
 - ◆ **Quanti anni? (1 anno = $3,15 \cdot 10^{10}$ msec) 315!**
- ◆ ...per programmi banali si arriva a tempi di esecuzione superiori al tempo passato dal big-bang

Terminazione del testing

- ◆ **Quando il programma si può ritenere analizzato a sufficienza?**
 - ◆ Criterio **temporale**: periodo di tempo predefinito
 - ◆ Criterio di **costo**: sforzo allocato predefinito
 - ◆ Criterio di **copertura**:
 - ◆ percentuale predefinita degli elementi di un modello del sw (si vedano i criteri di accettazione)
 - ◆ legato ad un criterio di selezione dei casi di test
 - ◆ Criterio **statistico**
 - ◆ MTBF (mean time between failures) predefinito (e confronto con un modello di affidabilità esistente)

- ◆ **Fault (Errore o Difetto interno):**
 - Un problema latente nel sistema, come un difetto nella progettazione, nel codice o nella configurazione.
 - È la causa potenziale di un comportamento non corretto, ma non genera effetti osservabili fino a quando non viene eseguito in determinate condizioni.
- ◆ **Esempio:** Un ciclo infinito in un algoritmo o una condizione errata (if ($a = b$) invece di if ($a == b$)).
- ◆ **Failure**
 - Il comportamento osservato del sistema che non corrisponde alle aspettative definite (specifiche o requisiti).
 - Si verifica quando un fault viene eseguito e porta a un risultato errato o a un comportamento imprevisto.
- ◆ **Esempio:** Un'applicazione restituisce un valore errato o si arresta.
- ◆ **Incident (Incidente):**
 - Una discrepanza osservata durante il testing o l'uso del sistema.
 - Non implica necessariamente che ci sia un failure: l'incidente potrebbe derivare da errori nei test stessi, errori nell'oracolo (risultati attesi errati), o inconsistenze nei requisiti.
- L'incident è documentato e analizzato per determinare se è effettivamente causato da un fault o da altri fattori.
- ◆ **Esempio:** Il sistema restituisce un risultato inatteso, ma potrebbe dipendere da un errore nei dati di test o nell'oracolo.

- ◆ **Testing (Test):**

Il testing consiste nell'esercitare il software utilizzando casi di test per identificare eventuali fault presenti nel sistema.

- ◆ **Test Case (Caso di Test):**

Un insieme di dati di input e risultati attesi (oracle) progettati per testare un componente con l'obiettivo di provocare failure e identificare eventuali fault.

- ◆ **Test Suite (Suite di Test):**

Un insieme di casi di test progettati per coprire una determinata funzionalità o un aspetto specifico del sistema.

Testing: Concetti

- ◆ Una **Componente** è una parte del sistema che può essere isolata per essere testata (un oggetto, un gruppo di oggetti, uno o più sottosistemi)
- ◆ Un **test case** è un insieme di input e di risultati attesi che esercitano una componente con lo scopo di causare fallimenti e rilevare fault.
 - ◆ **Ha 5 attributi: name, location, input, oracle, e log.**
 - ◆ **Name.** Per distinguere da altri test case (euristica: determinare il name a partire dal nome della componente o il requisito che si sta testando)
 - ◆ **Location.** Descrive dove il test case può essere trovato (un path name oppure un URL al programma da eseguire e il suo input)
 - ◆ **Input.** Descrive l'insieme di dati in input o comandi che l'attore del test case deve inserire
 - ◆ **Oracle.** Il comportamento atteso dal test case, ovvero la sequenza di dati in output o comandi che la corretta esecuzione del test dovrebbe far avere.
 - ◆ **Log.** È l'insieme delle correlazioni del comportamento osservato con il comportamento atteso per varie esecuzioni del test

Test case e relazioni

- ♦ 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 l'altro test case
- ♦ **Un buon modello di test deve contenere poche relazioni → velocizzare il processo di testing**
- ♦ Test case sono classificati in **blackbox** e **whitebox**.
 - ◆ **Blackbox.** Si focalizza sul comportamento I/O. Non si preoccupa della struttura interna della componente
 - ◆ **Whitebox.** Si focalizza sulla struttura interna della componente (non sul comportamento I/O): ogni stato nel modello dinamico dell'oggetto e ogni interazione tra gli oggetti viene testata.

Test stub and test driver

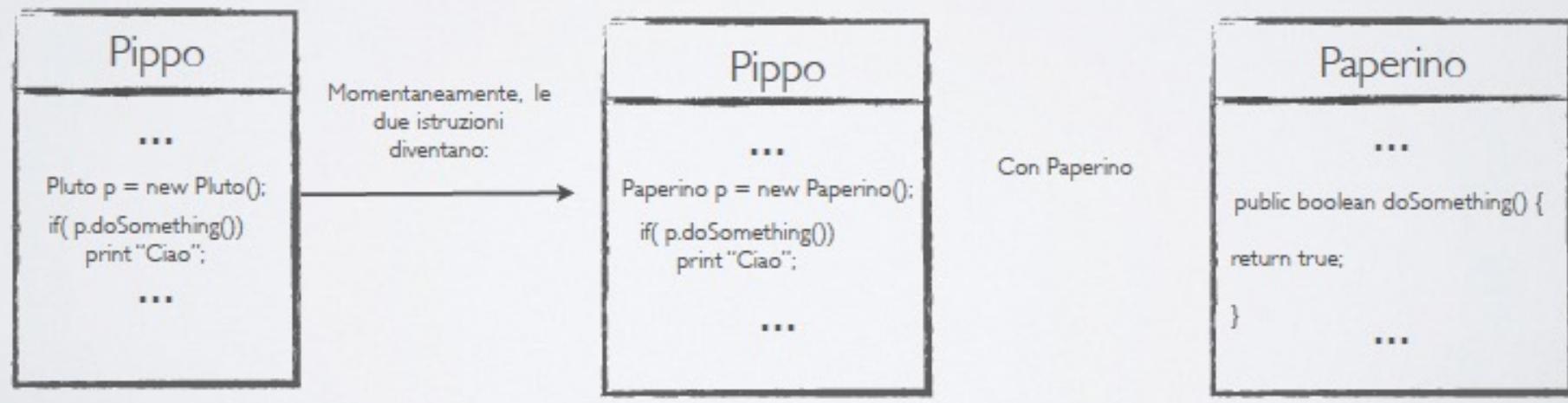
- ◆ Eseguire test case su una componente o una combinazione di componenti richiede che le componenti testate siano isolate dal resto del sistema
- ◆ **Test driver** e **test stub** sono usati per sostituire le parti mancanti del sistema
 - ◆ Un **test stub** è una implementazione parziale di componenti da cui la componente testata dipende (componenti che sono chiamate dalla componente testata).
 - ◆ Un **test driver** è una implementazione parziale di una componente che dipende dalla componente testata (componente che chiama la componente testata).

TEST STUB

Uno stub è una implementazione parziale di un componente da cui dipende il componente sotto test.

ESEMPIO:

Vogliamo testare Pippo.java. All'interno di Pippo c'è una chiamata ad una classe Pluto. La classe Pluto deve essere implementata da Veronica, la quale è in ritardo con la consegna. Possiamo comunque testare Pippo implementando uno stub, ovvero un componente che simuli il comportamento di Pluto.



TEST DRIVER

Un driver è una implementazione parziale di un componente che dipende dal componente sotto test.

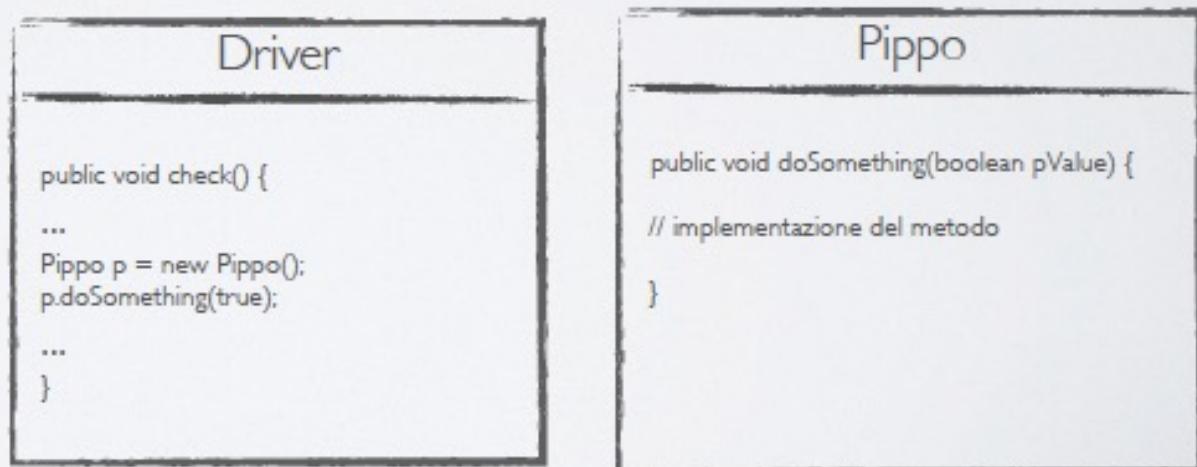
ESEMPIO:

Vogliamo testare Pippo.java. In particolare, il metodo doSomething() riceve come parametro dall'interfaccia grafica un valore booleano che serve per effettuare dei controlli all'interno del metodo.

PROBLEMA: Non si ha a disposizione l'interfaccia grafica!

Risolviamo implementando una classe Driver che chiama il metodo doSomething() passando il parametro necessario per il testing di Pippo.

Il vantaggio di avere un Driver è che per testare Pippo NON ABBIAMO BISOGNO DI IMPLEMENTARE UN'INTERAFACCIA GRAFICA.



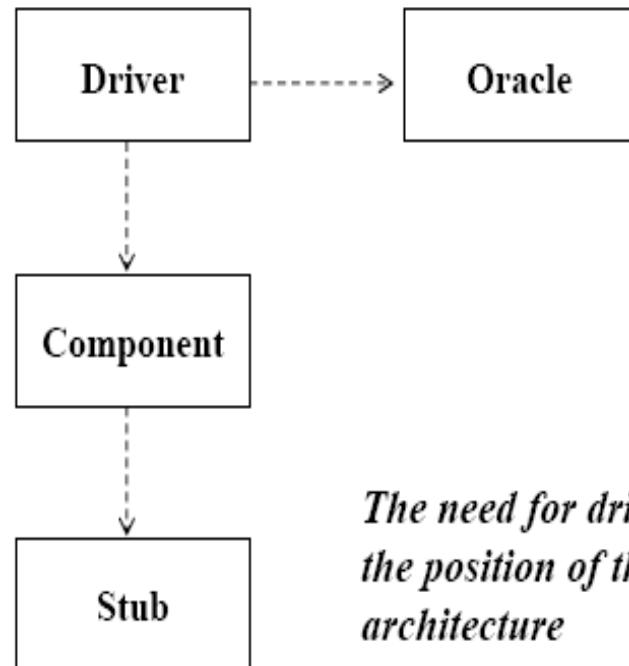
Test stub and test driver

- ◆ Un test stub deve fornire la stessa API del metodo della componente simulata e ritornare un valore il cui tipo è conforme con il tipo del valore di ritorno specificato nella signature.
 - ◆ Se l'interfaccia di una componente cambia anche il corrispondente test driver e test stub devono cambiare → gestione delle configurazioni
- ◆ L'implementazione di un test stub non è una cosa semplice.
 - ◆ Non è sufficiente scrivere un test stub che semplicemente stampa un messaggio attestante che il test stub è iniziato
 - ◆ La componente chiamata deve fare un qualche lavoro, se il test stub non simula il giusto comportamento la componente testata potrebbe avere un failure non perché c'è un fault
 - ◆ Il test stub non può restituire sempre lo stesso valore
 - ◆ Compromesso, tra implementare un test stub accurato e sostituire il test stub con la componente reale
 - ◆ spesso test stub e test driver sono scritti dopo che la componente è stata realizzata (purtroppo se si è in ritardo non vengono scritti...)

Il problema dello scaffolding

Driver, Stubs, and Scaffolding

The oracle knows the expected output for an input to the component to be compared with the actual output to identify failures

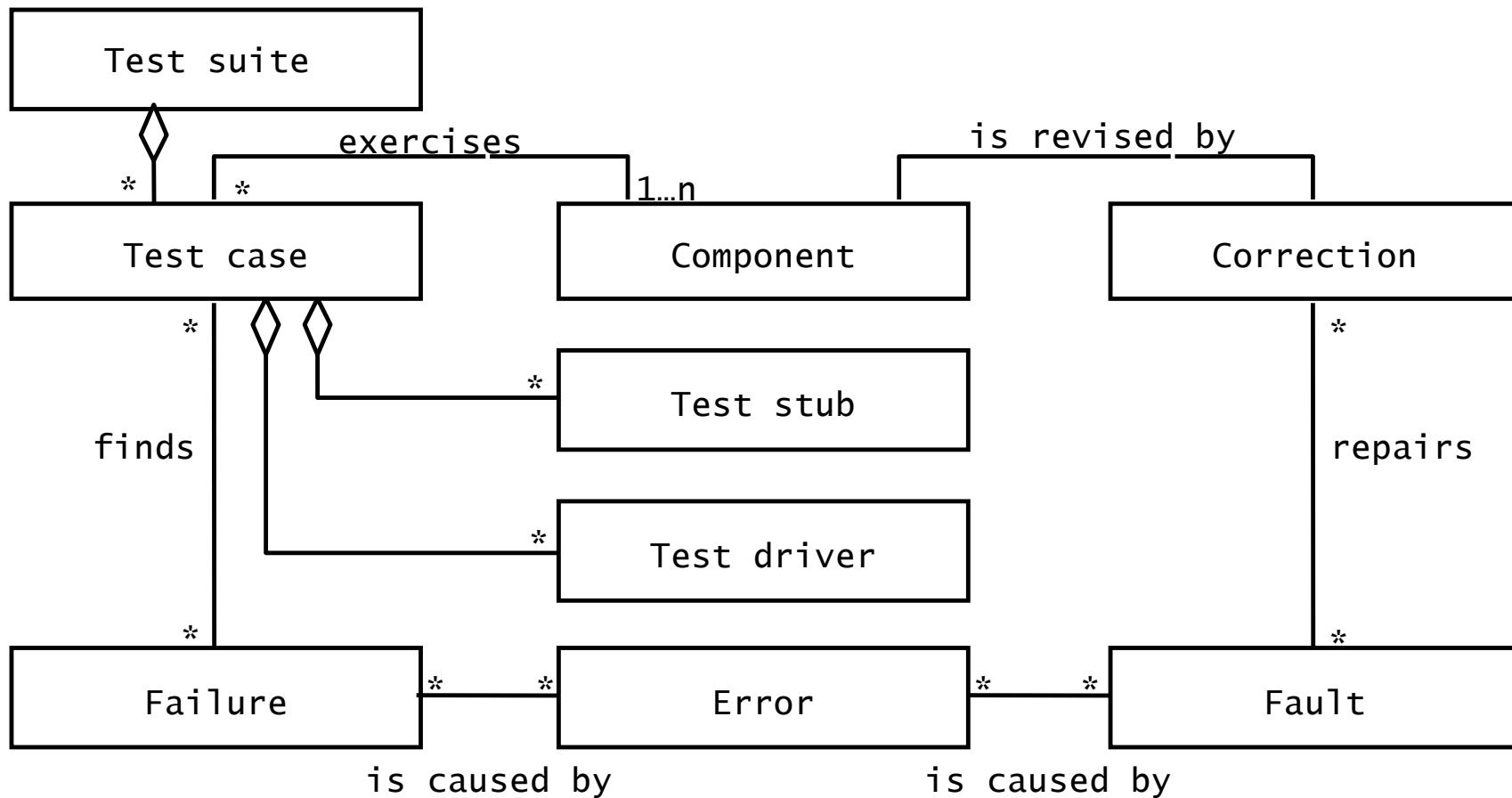


The need for drivers and stubs depends on the position of the component in the system architecture

Correzioni

- ◆ Quando i test sono stati eseguiti e i fallimenti sono stati rilevati, gli sviluppatori cambiano la componente per eliminare il fault sospettato
- ◆ Una **correzione** è un cambiamento di una componente. Lo scopo è riparare un fault.
- ◆ Una correzione potrebbe introdurre nuovi fallimenti. Molte tecniche possono essere usate per gestire tali difetti:
 - ◆ **Problem tracking.** Include la documentazione di ogni fallimento, stato di errore, e bug rilevato, la sua correzione, e la revisione delle componenti coinvolte nel cambiamento → consente di restringere la ricerca di nuovi fault
 - ◆ **Regression testing.** Riesecuzione dei test precedenti subito dopo il cambiamento. Per assicurare che le funzionalità che lavorano prima della correzione non sono state influenzate → particolarmente importante quando si utilizza un approccio iterativo allo sviluppo come nel caso OO
 - ◆ **Rational maintenance.** Include la documentazione delle motivazioni alla base dei cambiamenti, e le relazioni dietro le motivazioni nella revisione della componente → consente agli sviluppatori di evitare di introdurre nuovi fault analizzando le assunzioni utilizzate per la costruzione della componente

Sommario delle definizioni



Managing testing

- ◆ Come gestire le testing activity per minimizzare le risorse necessarie
 - ◆ Alla fine, gli sviluppatori dovrebbero rilevare e quindi riparare un numero sufficiente di bug tale che il sistema soddisfi i requisiti funzionali e non funzionali entro un limite accettabile da parte del cliente
- ◆ Pianificazione delle testing activity (diagramma di PERT che mostra le dipendenze)
- ◆ Le attività sono documentate in 4 tipi di documenti:
 - ◆ il **Test Plan** che si focalizza sugli aspetti manageriali;
 - ◆ Ogni test case è documentato attraverso un **Test Case Specification**;
 - ◆ Ogni esecuzione di un test case è documentata attraverso il **Test Incident Report**;
 - ◆ Il **Test Report Summary** elenca tutti i fallimenti rilevati durante i test che devono essere investigati

Documenting testing: Planning

- ◆ ***Test Plan:*** focuses on the managerial aspects of testing.
 - ◆ It documents the scope, the approach, resources, schedule of testing activities.
 - ◆ The requirements and the components to be tested are identified in this document
- ◆ ***Test Case Specification:*** documents each test
 - ◆ This document contains the inputs, drivers, stubs, and expected outputs of the tests, as well as the tasks to be performed

Test Plan

1. Introduction
2. Relationship to other documents
3. System overview
4. Features to be tested/not to be tested
5. Pass/Fail criteria
6. Approach
7. Testing materials
8. Test Cases
9. Testing Schedule

Esempi: [Test Plan Template](#); [esempio Test Plan](#)

Test Case Specification

1. Test case specification identifier
2. Test items
3. Input Specifications
4. Output Specifications
5. Environmental needs
6. Special procedural requirements
7. Intercase dependencies

IEEE Template IEEE 829-1998

Esempio: .doc, excel

Documenting testing: Execution Documents

◆ *Test Incident Report:*

- ◆ Per **ogni test fallito**, fornisce dettagli sul risultato effettivo rispetto a quello atteso, oltre ad altre informazioni utili a comprendere il motivo del fallimento del test.
 - ◆ Questo documento è deliberatamente chiamato "rapporto di incidente" e non "rapporto di errore".
 - La ragione è che una discrepanza tra risultati attesi ed effettivi può verificarsi per vari motivi diversi da un difetto nel sistema.
 - Questi motivi includono: risultati attesi errati, esecuzione scorretta del test o incoerenze nei requisiti che consentono più di un'interpretazione.

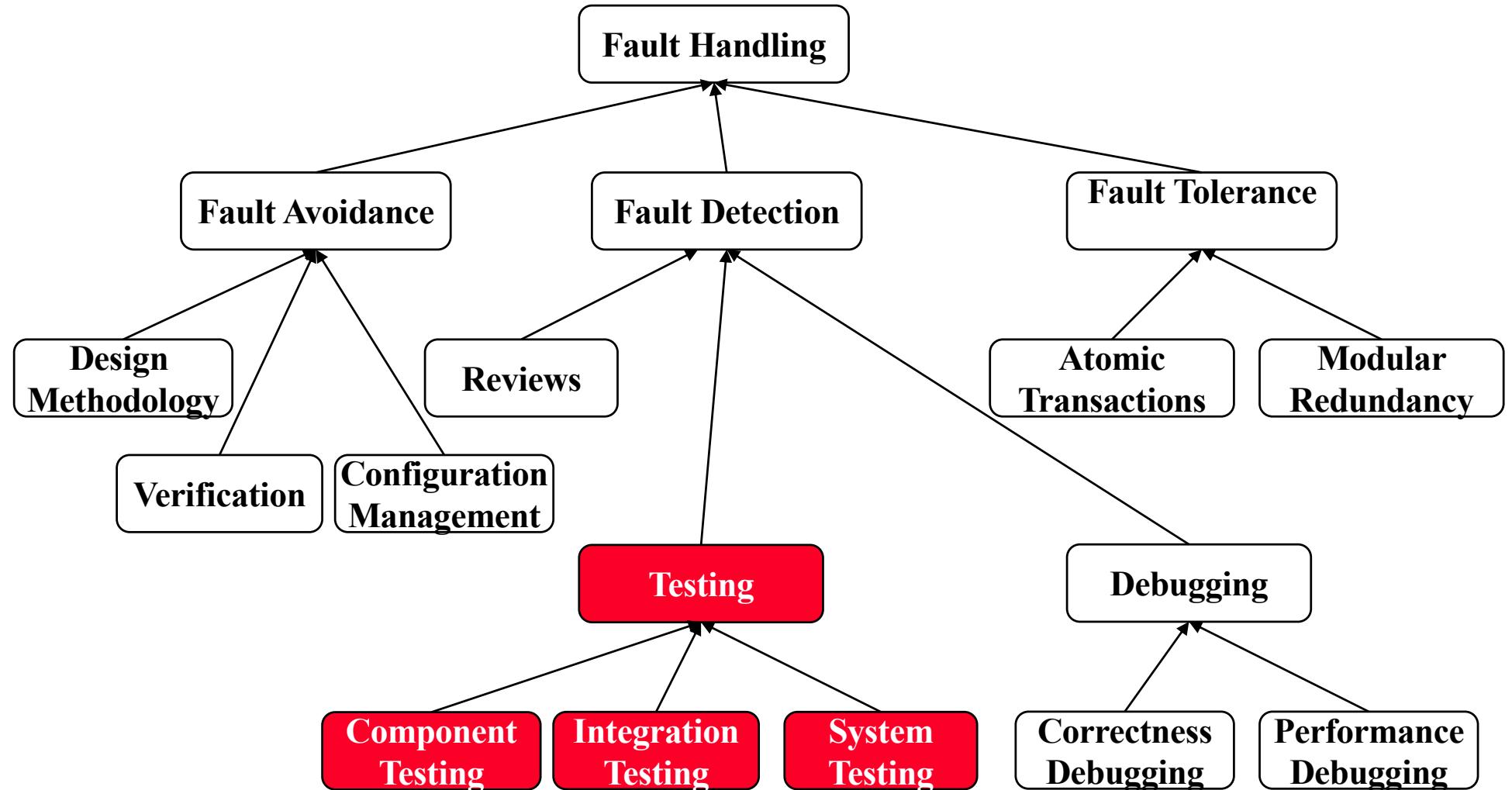
Documenting testing: Execution Documents

◆ *Test Summary Report:*

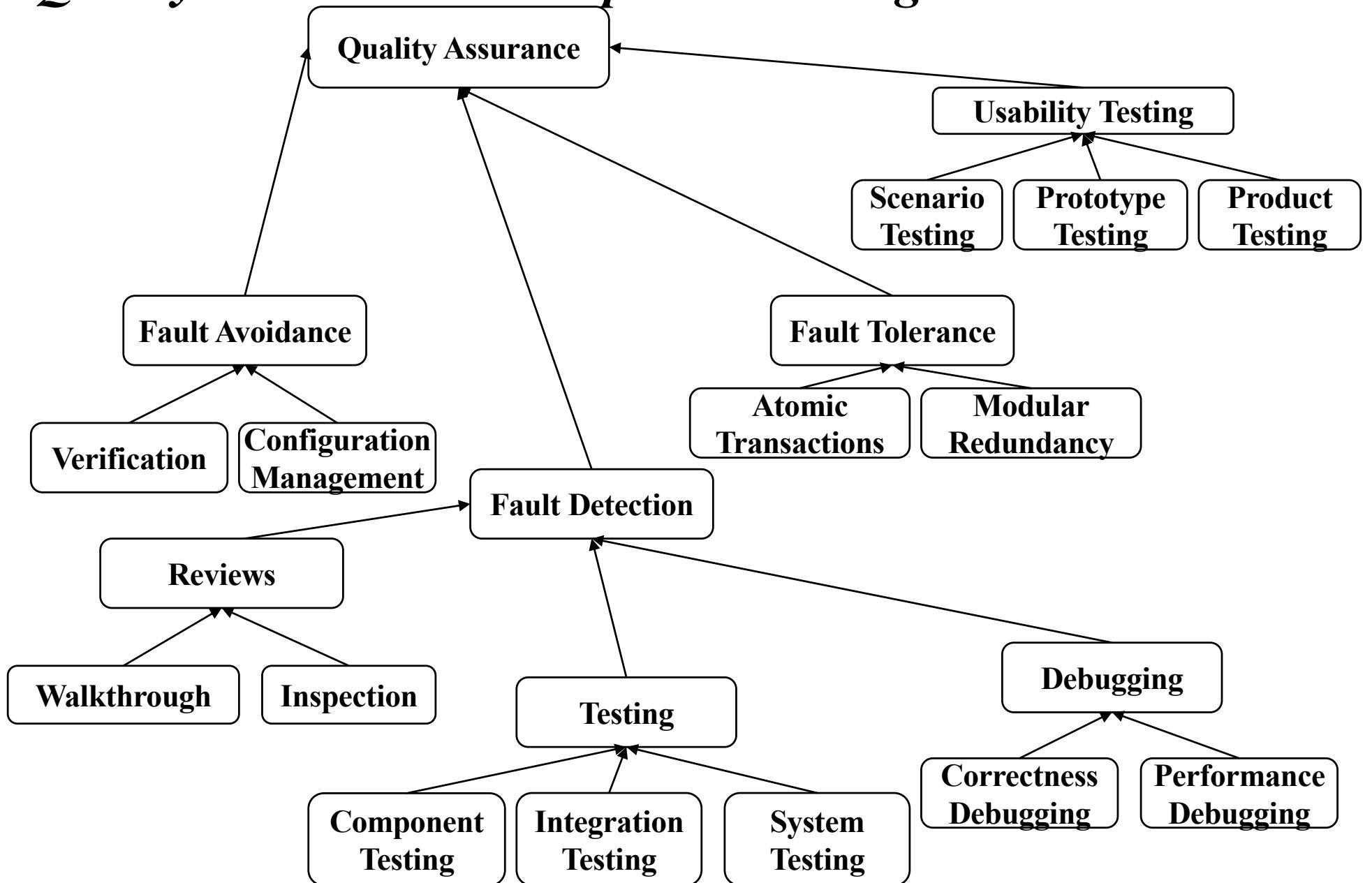
- ◆ **Fornisce uno stato generale del testing:**

Questo documento riassume lo stato complessivo del processo di testing, includendo i risultati, i test superati, quelli falliti e le eventuali problematiche identificate.

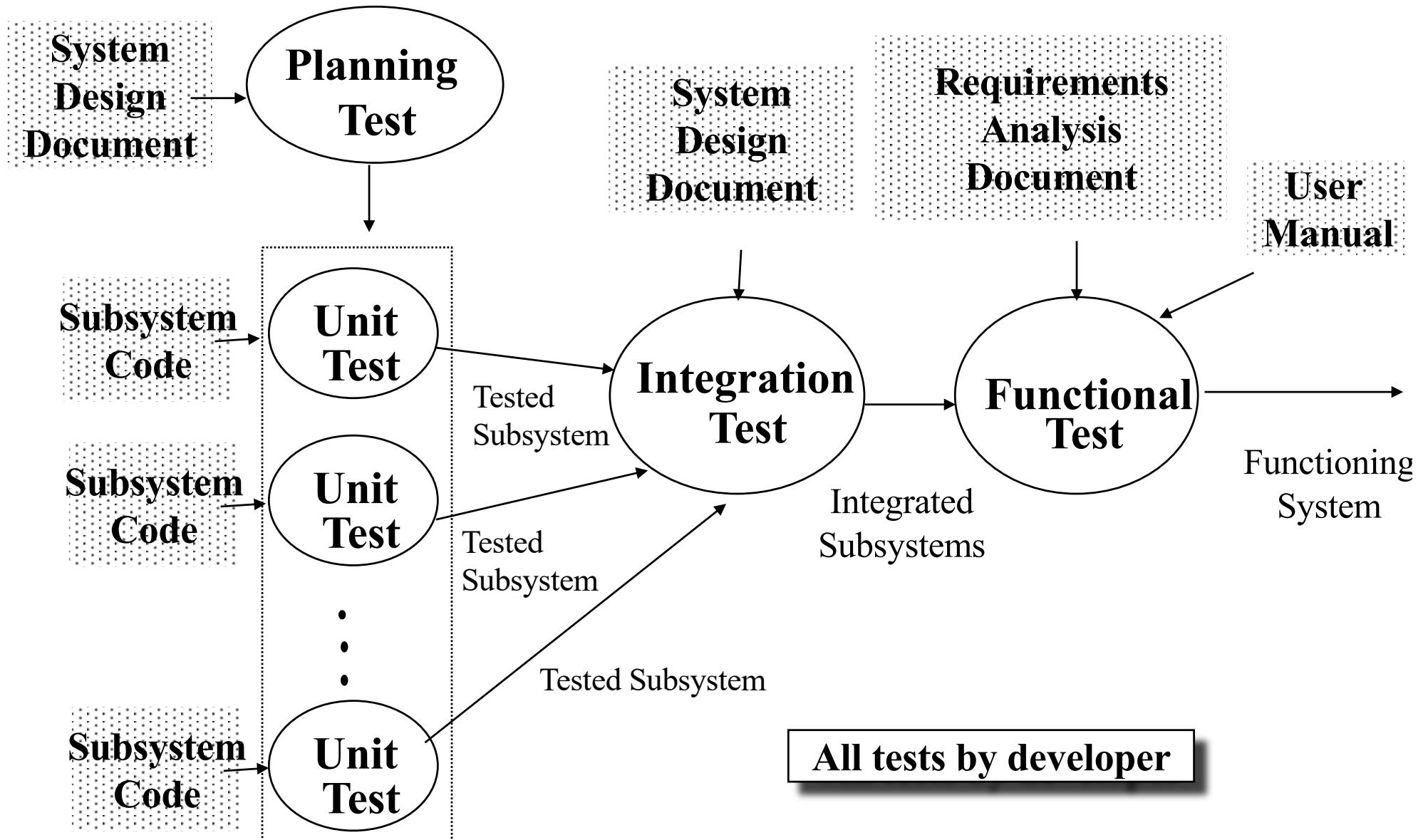
Fault Handling Techniques



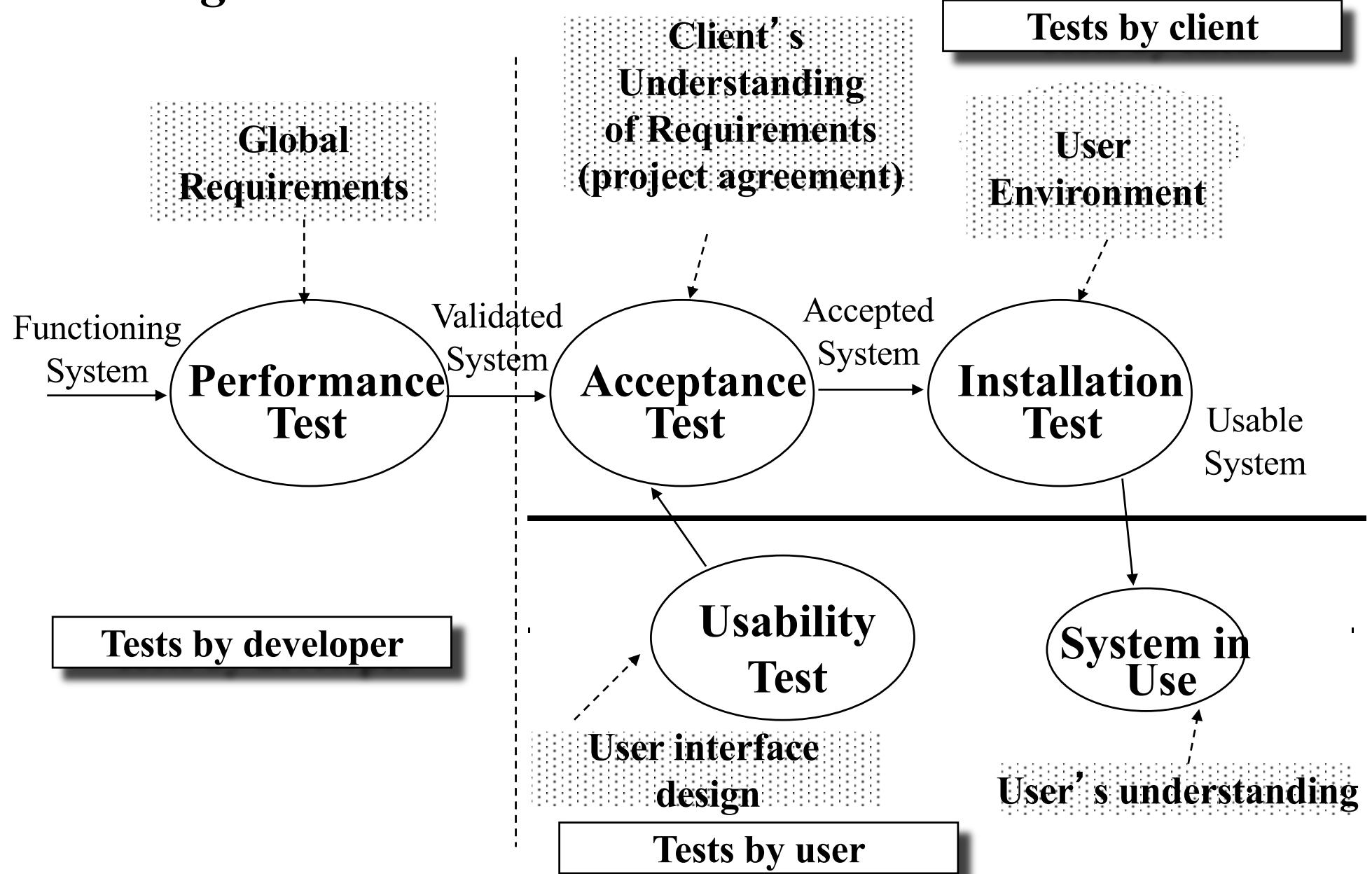
Quality Assurance encompasses Testing



Testing Activities



Testing Activities ctd



Types of Testing

- ◆ Unit Testing:
 - ◆ Individual subsystem
 - ◆ Carried out by developers
 - ◆ Goal: Confirm that **subsystem is correctly coded** and carries out the intended functionality
- ◆ Integration Testing:
 - ◆ Groups of subsystems (collection of classes) and eventually the entire system
 - ◆ Carried out by developers
 - ◆ Goal: Test the **interface** among the subsystems

System Testing

- ◆ System Testing:
 - ◆ The entire system
 - ◆ Carried out by developers
 - ◆ Goal: Determine if the system meets the requirements (functional and global)
- ◆ Acceptance Testing:
 - ◆ Evaluates the system delivered by developers
 - ◆ Carried out by the client. May involve executing typical transactions on site on a trial basis
 - ◆ Goal: Demonstrate that the system meets customer requirements and is ready to use
- ◆ Implementation (Coding) and testing go hand in hand

Attività di Testing

- ◆ **Component inspection.** Trova i fault in una componente individuale attraverso l’ispezione manuale del codice sorgente
- ◆ **Usability testing.** Trova le differenze tra il sistema e l’attesa dell’utente per quanto riguarda l’uso del sistema
- ◆ **Unit testing.** Trova fault isolando una componente individuale usando test stub e test driver, e esercitando la componente tramite un test case
- ◆ **Integration testing.** Trova fault integrando differenti componenti insieme
- ◆ **System testing.** Si focalizza sul sistema completo, i suoi requisiti funzionali e non funzionali, e il suo ambiente

Component Inspection

- ◆ Ispezioni trovano **fault** in una componente rivedendo il codice sorgente in meeting formali.
- ◆ Possono essere condotti prima o dopo il testing delle unità
- ◆ E' condotta da un team di sviluppatori, incluso l'autore della componente, un moderatore e uno o più revisori che trovano i bug nella componente
- ◆ Il metodo proposto da Fagan:
 - ◆ **Overview.** L'autore presenta l'obiettivo e lo scope della componente e i goal dell'ispezione
 - ◆ **Preparation.** I revisori analizzano l'implementazione della componente
 - ◆ **Inspection meeting.** Un reader legge il codice sorgente e spiega ciò che dovrebbe fare, il team di ispezione analizza la componente e evidenzia eventuali **fault**, il moderatore tiene traccia. La maggior parte del tempo è passata a discutere, ma le soluzioni per riparare il bug non sono esplorate in questo punto
 - ◆ **Rework.** L'autore rivede la componente
 - ◆ **Follow-up.** Il moderatore controlla la qualità del rework e determina la componente che deve essere ispezionata di nuovo.

Component Inspection

- ◆ Il metodo proposto da Fagan è percepito come time-consuming
- ◆ **Active Design Review:** processo di ispezione rivisitato proposto da Parnas:
 - ◆ **Preparation.** I revisori analizzano l'implementazione della componente e individuano i fault. Compilano un questionario che testa la loro comprensione della componente
 - ◆ **Non c'è Inspection meeting.** L'autore incontra separatamente ogni reviewer per collezionare feedback sulla componente
- ◆ Il metodi di ispezione sono molto efficaci: Circa 85% dei fault può essere individuato

Usability testing

- ◆ Per testare la comprensibilità del sistema da parte dell'utente
- ◆ I rappresentanti dei potenziali utenti trovano problemi “usando” le interfacce utente o una loro simulazione (possono coinvolgere anche dettagli delle interfacce – “look and feel”, layout geometrico delle schermate, la sequenza delle interazioni)
- ◆ Le tecniche sono basate sull’approccio degli esperimenti controllati:
 - ◆ Gli sviluppatori prima selezionano un insieme di obiettivi
 - ◆ Gli obiettivi sono poi valutati in una serie di esperimenti in cui viene chiesto ai partecipanti di eseguire una serie di attività (per esercitare le interfacce utente che si stanno investigando)
 - ◆ Gli sviluppatori osservano i partecipanti e raccolgono una serie di informazioni
 - ◆ “misurando” le performance degli utenti (per esempio, il tempo per realizzare le attività) e
 - ◆ le loro opinioni per identificare problemi specifici del sistema o collezionare idee per migliorarlo

Usability test types

- ◆ **Scenario test.** Viene presentato a uno o più utenti uno Scenario visionario.
- ◆ Gli sviluppatori determinano quanto velocemente gli utenti comprendono lo scenario, la bontà del modello e come reagiscono gli utenti alla descrizione del nuovo sistema.
- ◆ Lo scenario selezionato dovrebbe essere il più realistico possibile
 - ◆ Utilizzo di mock-up o di storyboard
 - ◆ Vantaggi: sono economici da realizzare e da ripetere
 - ◆ Svantaggi: gli utenti non possono interagire direttamente con il sistema, i dati sono fissi

Usability test types

- ◆ **Prototype test.** Agli utenti finali viene presentato una parte del software che implementa gli aspetti chiave del sistema
 - ◆ **Vertical prototype.** Implementa completamente uno use case. Prototipi funzionali sono usati per valutare le richieste cruciali
 - ◆ **Horizontal prototype.** Implementa un singolo layer nel sistema (per esempio un **user interface prototype** che rappresenta l'interfaccia per molti use case, senza fornire funzionalità)
 - ◆ **Vantaggi.** Forniscono una vista realistica del sistema all'utente e il prototipo può essere concepito per collezionare informazioni dettagliate
 - ◆ **Svantaggi.** Richiede un impegno maggiore nella costruzione rispetto agli scenari cartacei
- ◆ **Product test.** Simile al prototype test, eccetto per il fatto che viene utilizzata una versione funzionale del sistema. Il test può essere affrontato solo dopo che una buona parte del sistema è stata sviluppata → richiede che il sistema sia facilmente modificabile

Usability test – basic elements

- ♦ Tutte e tre le tecniche prevedono:
 - ◆ Sviluppo degli obiettivi del test (confronto tra due stili di interazione, qualche help necessario, quale tipo di training è richiesto, ..)
 - ◆ Selezionare un campione rappresentativo degli utenti finali
 - ◆ Una simulazione (o il reale) dell'ambiente di lavoro
 - ◆ Interrogazioni controllate ed estensive degli utenti alle prese con l'utilizzo del sistema attraverso i test
 - ◆ Collezione e analisi dei risultati qualitativi e quantitativi
 - ◆ Raccomandazioni su come migliorare il sistema

Unit testing

- ◆ Si focalizza sui building block del sistema (oggetti e sottosistemi)
- ◆ Motivazioni:
 - ◆ Si riduce la complessità concentrandosi su una sola unit
 - ◆ È più facile correggere i bug, poiché poche componenti sono coinvolte
 - ◆ Diverse unità sono testate in parallelo
- ◆ Le unità candidate per il test sono prese dal modello a oggetti e dalla decomposizione dei sottosistemi
- ◆ L'insieme minimale di oggetti: oggetti partecipanti negli use case
 - ◆ I sottosistemi devono essere testati dopo che ogni classe e oggetto del sottosistema è stato testato individualmente

Black-box Testing per Unit testing

- ◆ Si focalizza sul comportamento I/O. Non si preoccupa della struttura interna della componente
- ◆ Se per ogni input dato, l'output corrisponde a quello predetto allora il modulo supera il test.
 - ◆ È quasi sempre impossibile generare tutti i possibili input ("test case")
- ◆ **Equivalence testing.** Obiettivo: ridurre il numero di test case effettuando un partizionamento sulla base dell'equivalenza del comportamento:
 - ◆ Si dividono le condizioni di input in classi di equivalenza
 - ◆ Scegliere i test case per ogni classe di equivalenza.
- ◆ **Boundary testing.** È un caso speciale dell' equivalence testing: invece di selezionare gli elementi nella classe di equivalenza si selezionano elementi “ai confini” della classe

Test Black-box

- ◆ Test funzionale che usa la specifica per partizionare il dominio di input
- ◆ p. e., la specifica del metodo “roots” suggerisce di considerare 3 diversi casi in cui ci sono zero, una e due radici reali
 - Testare ogni “categoria”
 - Testare i confini tra le categorie

Nessuna garanzia, ma l’esperienza dimostra che spesso i malfunzionamenti sorgono ai “confini”

- ◆ Black-box *funzionale*
 - ◆ casi di test determinati in base a ciò che il componente deve fare, la sua *specifica*
- ◆ White-box *strutturale*
 - ◆ casi di test determinati in base a che come il componente è implementato, il codice

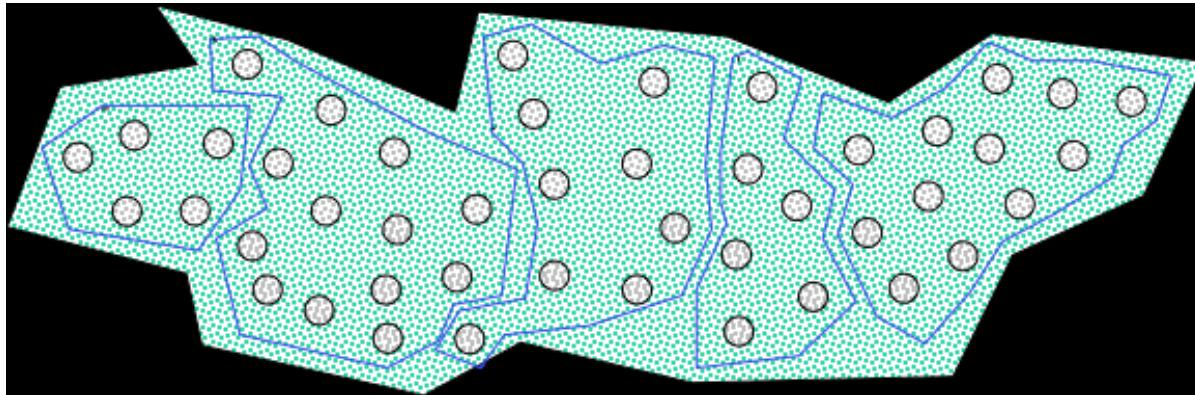
Utilità di test funzionali

- ◆ Non è necessario che esista il codice per determinare i dati di test
 - ◆ **basta la specifica--formale o informale**
 - ◆ **nel caso di *extreme programming* i test sono la specifica!**
- ◆ Questi possono dunque essere determinati in fase di progettazione
- ◆ Useremo esempi di programmi molto banali per vedere alcune tecniche

Black-box Testing per Unit testing

- ◆ Selezione delle classi di equivalenza (non ci sono regole, solo linee guida):
 - ◆ L'input è valido per un range di valori.
 - ◆ Seleziona i test case da 3 classi di equivalenza:
 - Sotto il range
 - dentro il range
 - sopra il range
 - ◆ L'input è valido se appartiene ad un insieme discreto.
 - ◆ Seleziona i test case da 2 classi di equivalenza :
 - Valore discreto valido
 - Valore discreto non valido

Partizionamento sistematico



- ◆ Si cerca di partizionare il dominio di input in modo tale che da tutti i punti del dominio ci si attende lo stesso comportamento (e quindi si possa prendere come rappresentativo un punto qualunque in esso)
 - ◆ L'esperienza dimostra poi che è anche opportuno prendere punti sui confini delle regioni
- ◆ Talvolta non è' una partizione in senso proprio (le classi di valori hanno intersezione non vuota)

Equivalence Class Testing

- ♦ **Motivazione:**

Vorremmo avere un senso di test completo e speriamo di evitare ridondanze

- **Classi di equivalenza:** partizionamento dell'insieme di input
 - **Copertura dell'intero insieme di input:** garantisce completezza
 - **Classi disgiunte:** evita la ridondanza
- **Casi di test:** un elemento per ogni classe di equivalenza
- **Attenzione:** le classi di equivalenza devono essere scelte con cura ...
- **Ipotesi:** immaginare il probabile comportamento sottostante ...

Weak/Strong Equ. Class Testing

- ◆ Three input variables of domains: A, B, C
- ◆ $A = A_1 \cup A_2 \cup A_3$ where $a_n \in A_n$
- ◆ $B = B_1 \cup B_2 \cup B_3 \cup B_4$ where $b_n \in B_n$
- ◆ $C = C_1 \cup C_2$ where $c_n \in C_n$
- ◆ Weak Equivalence Class Testing: choose one variable value from each equivalence class
- ◆ Strong Equivalence Class Testing: based on the cartesian product of the partition subsets ($A \times B \times C$), i.e., test all class interactions

WECT Test Cases

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

Quanti Test Cases?

*Dipende dal numero di partizioni per ogni variabile
...dalla cardinalità max*

- ♦ Svantaggio di Equivalence (e Boundary test): non sono esplorate le combinazioni di dati input al test.
 - ♦ **In molti casi è proprio una combinazione di certi valori a causare un bug**

SECT Test Cases

| 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 |

Quanti Test Cases?

NextDate Example

- ◆ NextDate è una funzione con tre variabili: mese, giorno, anno.
Restituisce la data del giorno successivo alla data di input.
Limite: 1812-2012
- ◆ Treatment Summary:
 - ◆ Se non è l'ultimo giorno del mese, la funzione NextDate si limiterà a incrementare il valore del giorno.
 - ◆ Alla fine di un mese, il giorno successivo sarà 1 e il mese verrà incrementato.
 - ◆ Alla fine dell'anno, sia il giorno che il mese vengono reimpostati a 1, e l'anno viene incrementato.
 - ◆ Infine, il problema degli anni bisestili rende interessante la determinazione della data del prossimo giorno.

- ◆ **Input del programma**
- **Mese:** da 1 a 12 (12 valori).
- **Giorno:** da 1 a 31 (31 valori), ma dipende dal mese (es., febbraio ha 28 o 29 giorni).
- **Anno:** limitato a un intervallo (1812–2012), quindi ci sono 201 anni possibili.
- ◆ **Combinazioni totali**
- ◆ Per un testing esaustivo, bisognerebbe verificare tutte le combinazioni possibili di mese, giorno e anno:
- ◆ Combinazioni possibili= $12 \times 31 \times 201 = 74.892$.
- ◆ Tuttavia, alcune combinazioni non sono valide (es., il 30 febbraio non esiste)

NextDate Equivalence Classes

- ◆ M1 = { month: month has 30 days}
- ◆ M2 = { month: month has 31 days}
- ◆ M3 = { month: month is February}
- ◆ D1 = {day: $1 \leq \text{day} \leq 28$ }
- ◆ D2 = {day: day = 29}
- ◆ D3 = {day: day = 30}
- ◆ D4 = {day: day = 31}
- ◆ Y1 = {year: year = 1900}
- ◆ Y2 = {year: $1812 \leq \text{year} \leq 2012$ AND $(\text{year} \neq 1900)$ AND $(\text{year mod } 4 = 0)$ }
- ◆ Y3 = {year: $(1812 \leq \text{year} \leq 2012)$ AND $\text{year mod } 4 \neq 0$)}

NextDate Test Cases (1)

- ◆ WECT: 4 test cases- maximum partition (D)

Case ID: Month, Day, Year, Output

WE1: 6, 14, 1900, 6/15/1900

WE2: 7, 29, 1912, 7/30/1912

WE3: 2, 30, 1913, Invalid Input

WE4: 6, 31, 1900, Invalid Input

- ◆ SECT: 36 test cases >> WECT

NextDate SECT: 36 test cases

| Case ID | Month | Day | Year | Expected Output |
|---------|-------|-----|------|-----------------|
| SE1 | 6 | 14 | 1900 | 6/15/1900 |
| SE2 | 6 | 14 | 1912 | 6/15/1912 |
| SE3 | 6 | 14 | 1913 | 6/15/1913 |
| SE4 | 6 | 29 | 1900 | 6/30/1900 |
| SE5 | 6 | 29 | 1912 | 6/30/1912 |
| SE6 | 6 | 29 | 1913 | 6/30/1913 |
| SE7 | 6 | 30 | 1900 | 7/1/1900 |
| SE8 | 6 | 30 | 1912 | 7/1/1912 |
| SE9 | 6 | 30 | 1913 | 7/1/1913 |
| SE10 | 6 | 31 | 1900 | ERROR |
| SE11 | 6 | 31 | 1912 | ERROR |
| SE12 | 6 | 31 | 1913 | ERROR |
| SE13 | 7 | 14 | 1900 | 7/15/1900 |
| SE14 | 7 | 14 | 1912 | 7/15/1912 |
| SE15 | 7 | 14 | 1913 | 7/15/1913 |
| SE16 | 7 | 29 | 1900 | 7/30/1900 |
| SE17 | 7 | 29 | 1912 | 7/30/1912 |

| | | | | |
|------|---|----|------|-----------|
| SE18 | 7 | 29 | 1913 | 7/30/1913 |
| SE19 | 7 | 30 | 1900 | 7/31/1900 |
| SE20 | 7 | 30 | 1912 | 7/31/1912 |
| SE21 | 7 | 30 | 1913 | 7/31/1913 |
| SE22 | 7 | 31 | 1900 | 8/1/1900 |
| SE23 | 7 | 31 | 1912 | 8/1/1912 |
| SE24 | 7 | 31 | 1913 | 8/1/1913 |
| SE25 | 2 | 14 | 1900 | 2/15/1900 |
| SE26 | 2 | 14 | 1912 | 2/15/1912 |
| SE27 | 2 | 14 | 1913 | 2/15/1913 |
| SE28 | 2 | 29 | 1900 | ERROR |
| SE29 | 2 | 29 | 1912 | 3/1/1912 |
| SE30 | 2 | 29 | 1913 | ERROR |
| SE31 | 2 | 30 | 1900 | ERROR |
| SE32 | 2 | 30 | 1912 | ERROR |
| SE33 | 2 | 30 | 1913 | ERROR |
| SE34 | 2 | 31 | 1900 | ERROR |
| SE35 | 2 | 31 | 1912 | ERROR |
| SE36 | 2 | 31 | 1913 | ERROR |

Discussione

- ◆ SECT è molto più «accurato» di WECT

Il SECT presuppone che le variabili siano indipendenti – le dipendenze genereranno casi di test "errori".

- ◆ Possibilmente troppi casi di test...

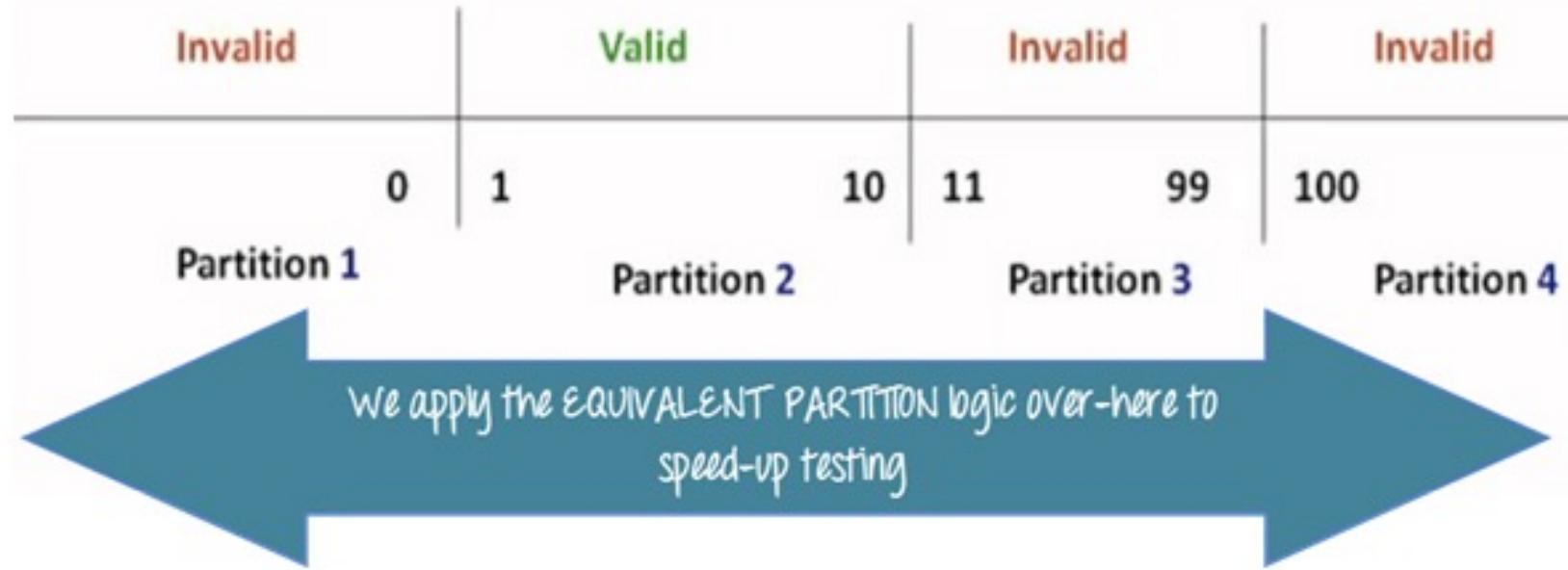
Boundary value testing: Motivazioni

- ◆ Abbiamo suddiviso i domini di input in classi, assumendo che il comportamento del programma sia «simile» per i dati nella stessa classe.
 - ◆ Alcuni errori tipici di programmazione tendono a verificarsi ai **confini** tra classi diverse.
 - ◆ È su questo che si concentra il testing dei valori limite (*boundary value testing*).
 - ◆ **complementare** alle tecniche precedenti

Equivalence Class Partitioning

- ◆ Può essere applicato a tutti i livelli del testing del software, come unità, integrazione, sistema, ecc.
- ◆ In questa tecnica, i dati di input vengono suddivisi in partizioni equivalenti che possono essere utilizzate per derivare casi di test, riducendo il tempo necessario per il testing grazie al numero ridotto di casi di test.
- ◆ Divide i dati di input del software in diverse classi di dati equivalenti.
- ◆ Puoi applicare questa tecnica quando c'è un intervallo nel campo di input.

- ◆ **Example 1: Equivalence and Boundary Value**
- ◆ Let's consider the behavior of Order Pizza Text Box Below
- ◆ Pizza values 1 to 10 is considered valid. A success message is shown.
- ◆ While value 11 to 99 are considered invalid for order and an error message will appear, "**Only 10 Pizza can be ordered**"
- ◆ **Here is the test condition**
 - ◆ Any Number greater than 10 entered in the Order Pizza field(let say 11) is considered invalid.
 - ◆ Any Number less than 1 that is 0 or below, then it is considered invalid.
 - ◆ Numbers 1 to 10 are considered valid
 - ◆ Any 3 Digit Number say -100 is invalid.
 - ◆ We cannot test all the possible values because if done, the number of test cases will be more than 100. To address this problem, we use equivalence partitioning hypothesis where we divide the possible values of tickets into groups or sets as shown below where the system behavior can be considered the same.



- ◆ The divided sets are called Equivalence Partitions or Equivalence Classes. Then we pick only one value from each partition for testing. The hypothesis behind this technique is **that if one condition/value in a partition passes all others will also pass**. Likewise, **if one condition in a partition fails, all other conditions in that partition will fail**.

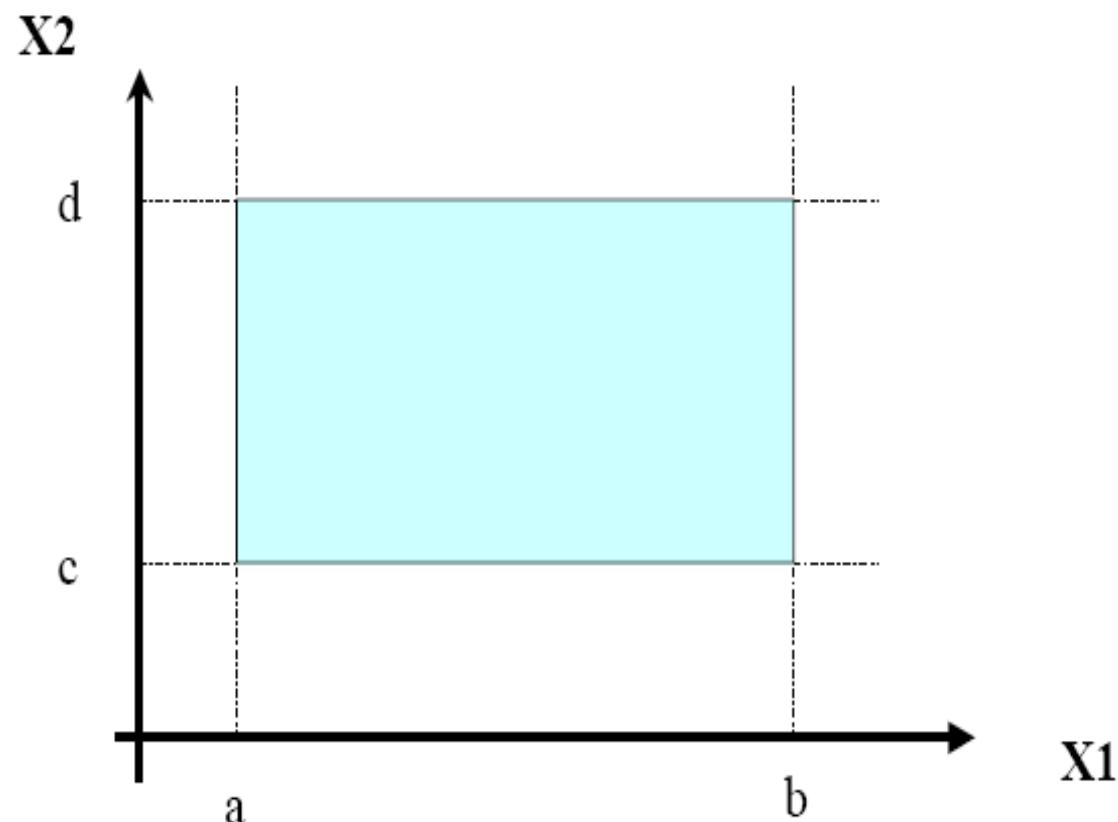
Boundary Value Analysis (BVA)

- ◆ Supponiamo una funzione F con due variabili, x1 e x2.
- ◆ Limiti : $a \leq x_1 \leq b$, $c \leq x_2 \leq d$.
- ◆ BVA Si concentra sui confini dello spazio di input per identificare i casi di test.
- ◆ La logica alla base è che gli errori tendono a verificarsi vicino ai valori estremi delle variabili di input, come dimostrato da alcuni studi.

Idea di base

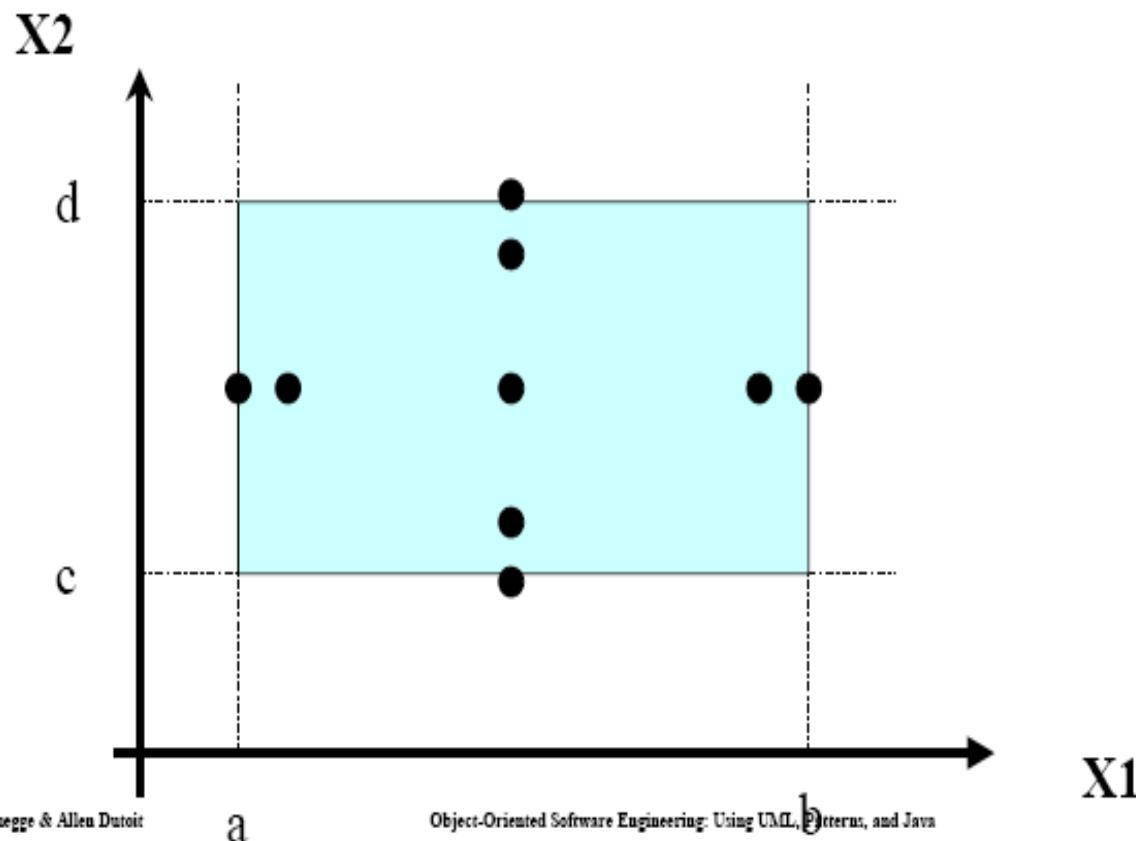
- ◆ I valori delle variabili di input vengono testati al loro minimo, appena sopra il minimo, a un valore nominale, appena sotto il massimo e al massimo.
- ◆ Convenzione: **min**, **min+**, **nom**, **max-**, **max**.
- ◆ **Si mantengono i valori di tutte le variabili tranne una al loro valore nominale**, lasciando che **una** variabile assuma il suo valore **estremo**.

Input Domain of Function F



Boundary Analysis Test Cases

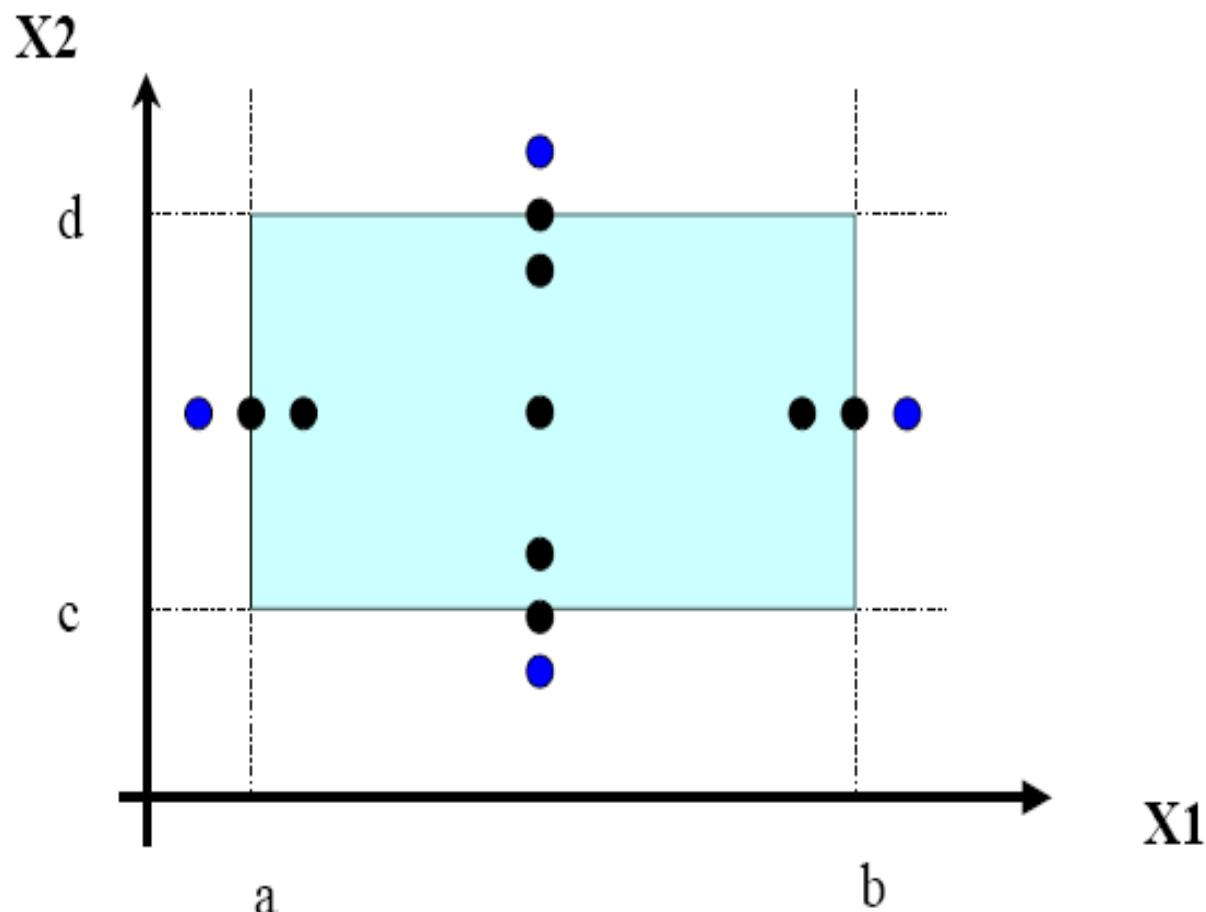
- ◆ Test set = { $\langle x_1_{\text{nom}}, x_2_{\text{min}} \rangle, \langle x_1_{\text{nom}}, x_2_{\text{min+}} \rangle, \langle x_1_{\text{nom}}, x_2_{\text{nom}} \rangle, \langle x_1_{\text{nom}}, x_2_{\text{max-}} \rangle, \langle x_1_{\text{nom}}, x_2_{\text{max}} \rangle, \langle x_1_{\text{min}}, x_2_{\text{nom}} \rangle, \langle x_1_{\text{min+}}, x_2_{\text{nom}} \rangle, \langle x_1_{\text{max-}}, x_2_{\text{nom}} \rangle, \langle x_1_{\text{max}}, x_2_{\text{nom}} \rangle \}$



Caso Generale e Limitazioni

- ◆ Una funzione con n variabili richiederà $4n+1$ casi di test.
- ◆ Funziona bene con variabili che rappresentano quantità fisiche limitate.
- ◆ Non considera la natura della funzione né il significato delle variabili.
- ◆ Tecnica elementare, adatta al testing di **robustezza** del sistema contro input non validi, particolarmente importante nei sistemi critici.

Robustness Testing



Esempio

- ◆ Funzione $\text{price}(\text{age}, \text{distance})$
 - **Variabili:**
 - **age:** 0–120
 - **distance:** 1–1.000
 - ◆ Il BVT testa ogni variabile ai suoi **limiti validi** mantenendo le altre variabili a un valore nominale.
1. **Casi di test richiesti:**
 1. Test **age** ai suoi limiti, con **distance** a 500 (**nominale**):
 $(\text{age}, \text{distance}): (0, 500), (1, 500), (50, 500), (119, 500), (120, 500)$
 2. Test **distance** ai suoi limiti, con **age** a 50 (**nominale**):
 $(\text{age}, \text{distance}): (50, 1), (50, 2), (50, 500), (50, 999), (50, 1,000)$
 - ◆ **Totale:** $5+5=10$ casi di test.

Esempio Robustness Testing (RT)

- ♦ Il RT estende il BVT per includere **valori non validi**.

1. Valori aggiunti:

1. age: **-1 (min-)**, 0 (min), 1 (min+), 50 (nom), 119 (max-), 120 (max), **121 (max+)**.
2. distance: **0 (min-)**, 1 (min), 2 (min+), 500 (nom), 999 (max-), 1,000 (max), **1.001 (max+)**.

2. Casi di test richiesti:

1. age con distance=500:

(-1,500), (0,500), (1,500),..., (121,500) → 7 casi.

2. distance con age=50:

(50,0), (50,1),..., (50,1.001) → 7 casi.

- ♦ **Totale:** 7+7=14 casi di test.

- ◆ **BVT:** Efficace per errori semplici ai confini del dominio valido.
- ◆ **RT:** Utile per verificare la robustezza del sistema contro input non validi, particolarmente importante nei sistemi critici.

| Caratteristica | Boundary Value Testing (BVT) | Robustness Testing |
|-------------------------------------|----------------------------------|---|
| Scopo | Testare i limiti validi | Testare i limiti validi e non validi |
| Input non validi | Non considerati | Inclusi |
| Numero di test case | Inferiore | Superiore (per i casi fuori dominio) |
| Applicazione principale | Garantire funzionalità al limite | Testare la capacità di gestire errori |
| Esempio di valori aggiuntivi | Nessuno | min-, max+ |

Worst Case Testing (WCT)

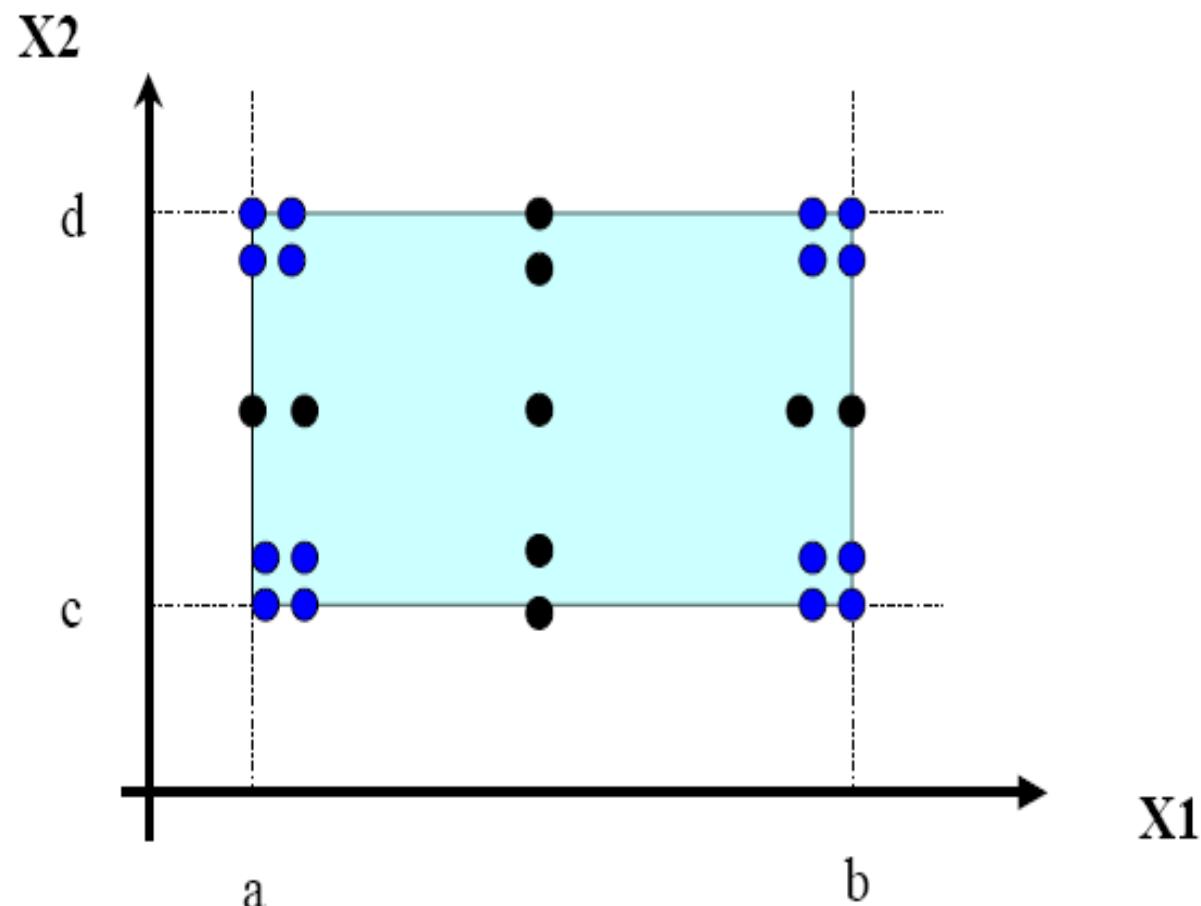
Il testing dei valori limite (*Boundary Value Testing*) parte dal presupposto che un malfunzionamento (failure), nella maggior parte dei casi, abbia origine da un unico errore (fault).

- ◆ Cosa succede quando più di una variabile assume un valore estremo?
- ◆ L'idea proviene dall'elettronica e dall'analisi dei circuiti.
- ◆ Prodotto cartesiano di {min, min+, nom, max-, max}.
- ◆ Chiaramente più completo rispetto all'analisi dei valori limite, ma molto più impegnativo: richiede 5^n casi di test.
- ◆ Strategia utile quando le variabili fisiche hanno molte interazioni e quando un malfunzionamento è costoso.
- ◆ Un passo ulteriore: **Robust Worst Case Testing**.

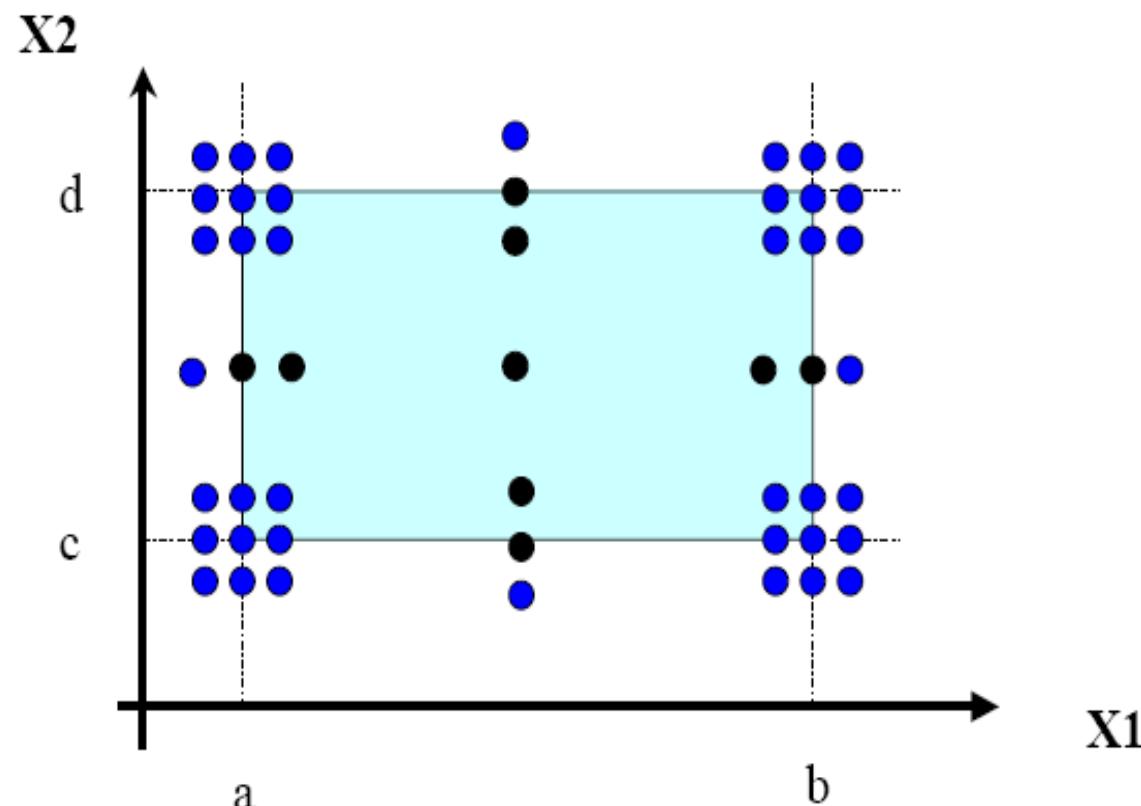
Worst Case Testing (WCT)

- ♦ è una tecnica di testing del software che si concentra sul valutare il comportamento di un programma nelle combinazioni di input che rappresentano i peggiori scenari possibili. Questo approccio viene utilizzato per assicurarsi che il sistema funzioni correttamente anche nelle condizioni più estreme o stressanti.
- ♦ **Caratteristiche principali del Worst Case Testing**
 1. **Massima combinazione di estremi di input:**
 1. Viene testato l'effetto di combinazioni di input estremi (minimi, massimi e altri valori al limite) su tutte le variabili di input contemporaneamente.
 2. Ogni variabile viene testata in combinazione con tutti i valori estremi delle altre variabili.
 3. Sistemi critici in cui è fondamentale verificare il comportamento in condizioni estreme (es. software di controllo di un reattore nucleare o sistemi aerospaziali). Quando il costo di un errore è molto elevato e vale la pena investire nel testing esteso.

WCT for 2 variables



Robust WCT for 2 variables



- ♦ Il WCT testa tutte le **combinazioni** di valori **limite validi** per ogni variabile.

1. Valori limite per age e distance:

1. **age:** 0 (min), 1 (min+), 50 (nom), 119 (max-), 120 (max)
2. **distance:** 1 (min), 2 (min+), 500 (nom), 999 (max-), 1.000 (max)

2. Combinazioni possibili:

1. Tutti i valori di **age** combinati con tutti i valori di **distance** → $5 \times 5 = 25$ casi di test.

♦ Esempio di test cases per WCT:

- (age, distance):
 - (0, 1), (0, 2), (0, 500), (0, 999), (0, 1.000)
 - (1, 1), (1, 2), ..., (1, 1.000)
 - ...
 - (120, 1), (120, 2), ..., (120, 1.000)

Worst Case Testing (WCT) con robustezza

- ◆ Il **Worst Case Testing (WCT)** con **robustezza** estende il concetto di WCT includendo **valori non validi** o al di fuori dei confini accettabili per ciascuna variabile.
- ◆ Questo approccio è particolarmente utile per sistemi critici, dove la gestione di input non validi può prevenire guasti o malfunzionamenti.

- ♦ **Definizione di valori testati:**

Per ogni variabile, consideriamo i seguenti valori:

- age: -1 (min-), 0 (min), 1 (min+), 50 (nom), 119 (max-), 120 (max), 121 (max+).
- distance: 0 (min-), 1 (min), 2 (min+), 500 (nom), 999 (max-), 1.000 (max), 1.001 (max+).

- ♦ **Combinazioni di valori:**

Il WCT con robustezza genera il **prodotto cartesiano** di tutti i valori delle due variabili:

- Casi di test totali = (7 valori per age) \times (7 valori per distance)
= 49 casi di test

- **Combinazioni di valori estremi:**

- (-1, 0), (-1, 1), ..., (-1, 1.001)
- (0, 0), (0, 1), ..., (0, 1.001)
- ...
- (121, 0), (121, 1), ..., (121, 1.001)

WCT con test di robustezza

- ◆ **Vantaggi:**

- **Massimizza la copertura, includendo tutte le combinazioni di input validi e non validi.**
- **Utile per sistemi critici, dove input fuori dal dominio possono causare errori gravi.**

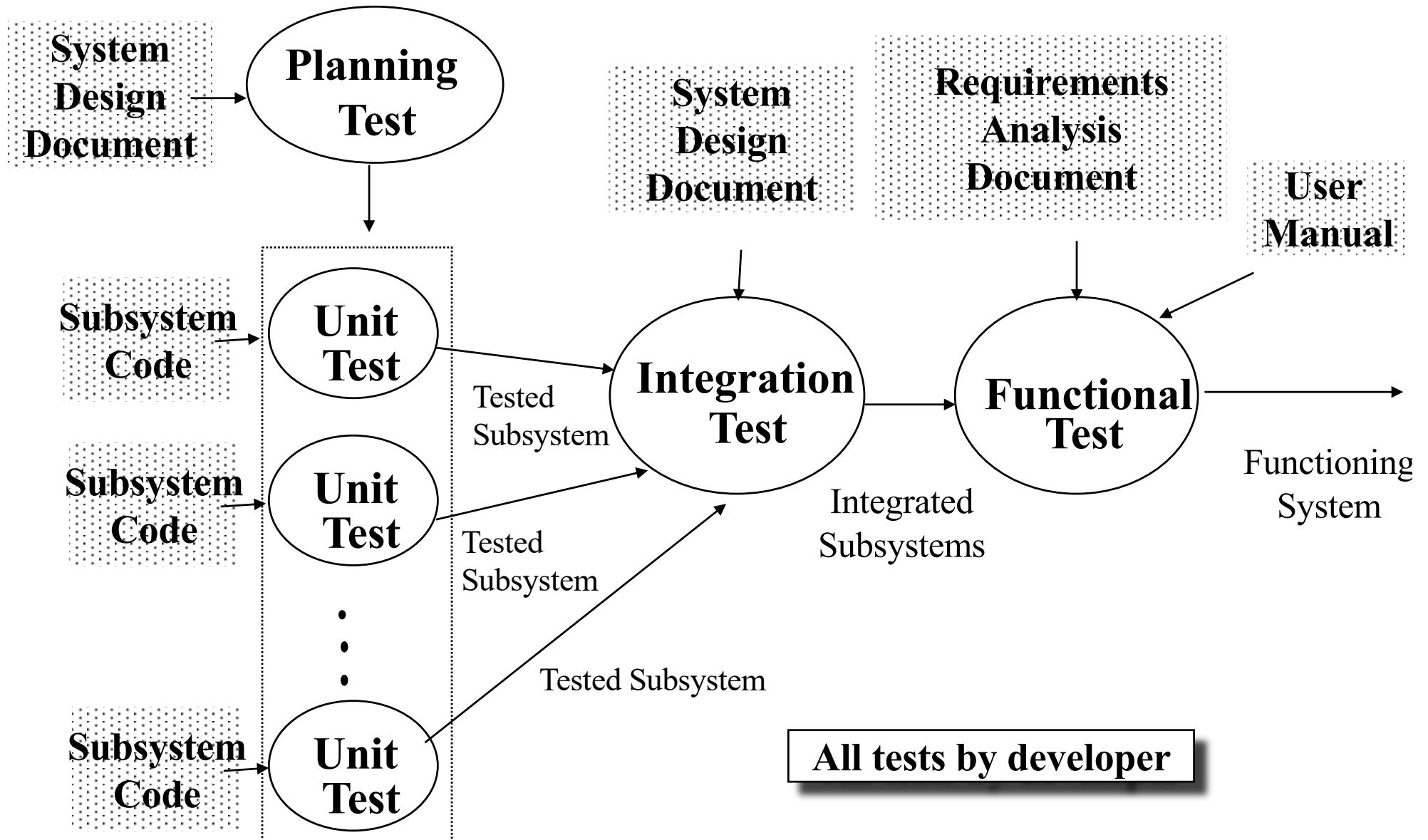
- ◆ **Limiti:**

- **Numero di casi di test molto elevato (7^n per n variabili).**
- **Richiede molte risorse, soprattutto per sistemi con più variabili o intervalli ampi.**
- ◆ Questo approccio è quindi indicato per applicazioni in cui i costi di errore sono alti (es. aerospaziale, medico).

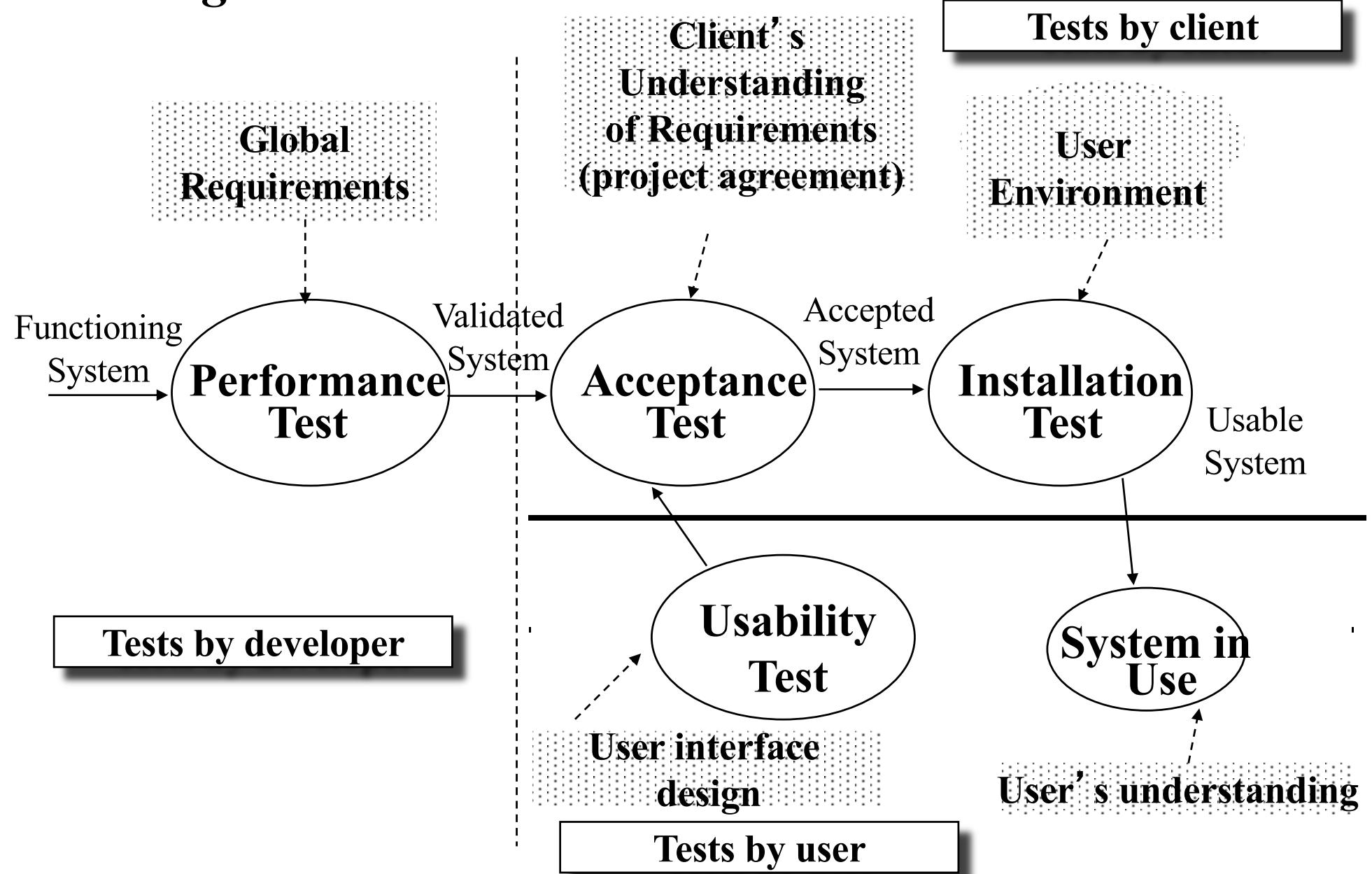
Confronto

| Metodo | Casi di test | Valori testati |
|----------------------------|--------------|--|
| BVT | 10 | Limiti validi |
| WCT | 25 | Tutte le combinazioni di limiti validi |
| Robustness Testing (RT) | 14 | Limiti validi + valori non validi (singolarmente) |
| WCT con Robustezza | 49 | Combinazioni di limiti validi e non validi |

Testing Activities



Testing Activities ctd



White-box Testing

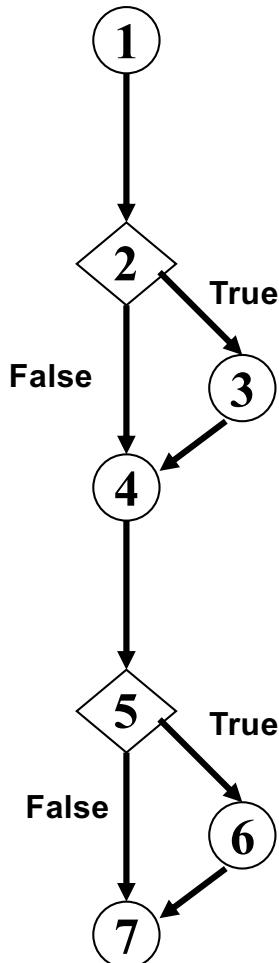
- ◆ Si focalizza sulla struttura interna della componente. Indipendentemente dall' input, ogni stato nel modello dinamico dell' oggetto e ogni integrazione tra gli oggetti viene testata.
 - ◆ **Completezza.** Es. Ogni statement nella componente è eseguito almeno una volta.
- ◆ 4 tipi di white-box testing
 - ◆ **Statement Testing**
 - ◆ **Loop Testing**
 - ◆ **Path Testing**
 - ◆ **Branch Testing**

White-box Testing

- ◆ Statement Testing (Algebraic Testing)
 - ◆ Si testano i singoli statement
- ◆ Loop Testing: definire in casi di test in modo da assicurarsi che:
 - ◆ Il loop deve essere saltato completamente.
 - ◆ Loop sia eseguito esattamente una volta
 - ◆ Loop da eseguire più di una volta
- ◆ Path testing:
 - ◆ Assicurarsi che tutti i path nel programma siano eseguiti
- ◆ Branch Testing (Conditional Testing)
 - ◆ assicurarsi che ogni possibile uscita da una condizione sia testata almeno una volta

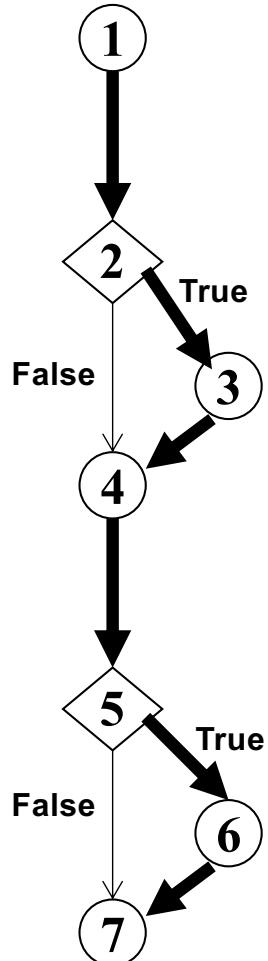
```
if ( i =TRUE) printf("YES\n");else printf("NO\n");  
Test cases: 1) i = TRUE; 2) i = FALSE
```

Program Control Graph of computeFine()

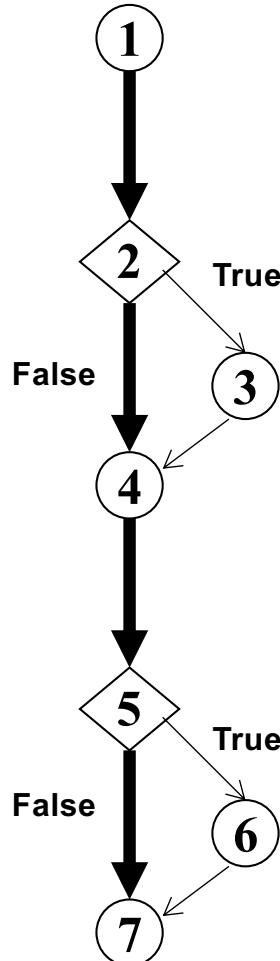


Distinct Paths of computeFine()

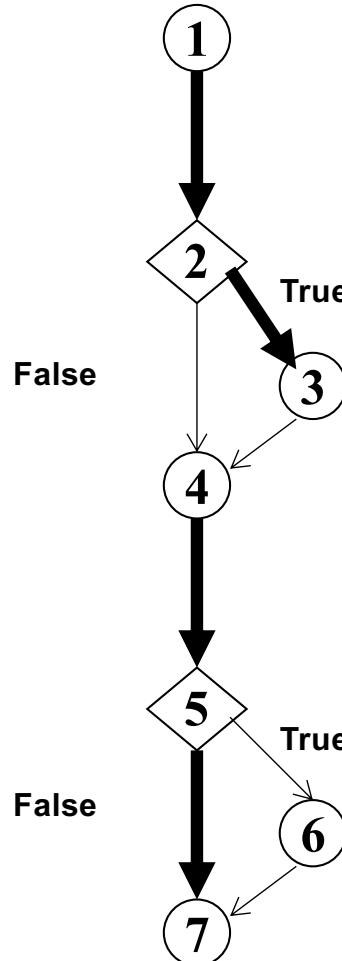
Path #1



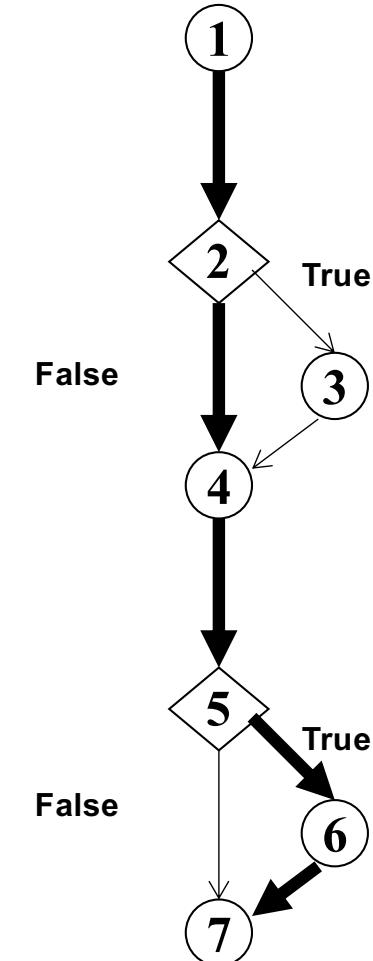
Path #2



Path #3



Path #4



Test Cases for Distinct Paths of computeFine()

| Test Case # | daysLate | printON | Path |
|--------------------|-----------------|----------------|---------------|
| 1 | 1 | TRUE | 1-2-3-4-5-6 7 |
| 2 | 60 | FALSE | 1-2-4-5--7 |
| 3 | 1 | FALSE | 1-2-3-4-5-7 |
| 4 | 60 | TRUE | 1-2-4-5-6-7 |

White-box Testing Example

```
FindMean(float Mean, FILE ScoreFile)
{ SumOfScores = 0.0; NumberOfScores = 0; Mean = 0;
  Read(ScoreFile, Score); /*Read in and sum the scores*/
  while (! EOF(ScoreFile) {
    if ( Score > 0.0 ) {
      SumOfScores = SumOfScores + Score;
      NumberOfScores++;
    }
    Read(ScoreFile, Score);
  }
  /* Compute the mean and print the result */
  if (NumberOfScores > 0 ) {
    Mean = SumOfScores/NumberOfScores;
    printf("The mean score is %f \n", Mean);
  } else
    printf("No scores found in file\n");
}
```

White-box Testing Example: Determining the Paths

```
FindMean (FILE ScoreFile)
```

```
{   float SumOfScores = 0.0;
```

```
    int NumberOfScores = 0;
```

```
    float Mean=0.0; float Score;
```

```
    Read(ScoreFile, Score);
```

```
    2 while (! EOF(ScoreFile) {
```

```
        3 if (Score > 0.0 ) {
```

```
            SumOfScores = SumOfScores + Score;
```

```
            NumberOfScores++;
```

```
        5 }
```

```
        Read(ScoreFile, Score);
```

```
    }
```

```
    /* Compute the mean and print the result */
```

```
    7 if (NumberOfScores > 0) {
```

```
        Mean = SumOfScores / NumberOfScores;
```

```
        printf(" The mean score is %f\n", Mean);
```

```
    } else
```

```
        printf ("No scores found in file\n");
```

```
}
```

1

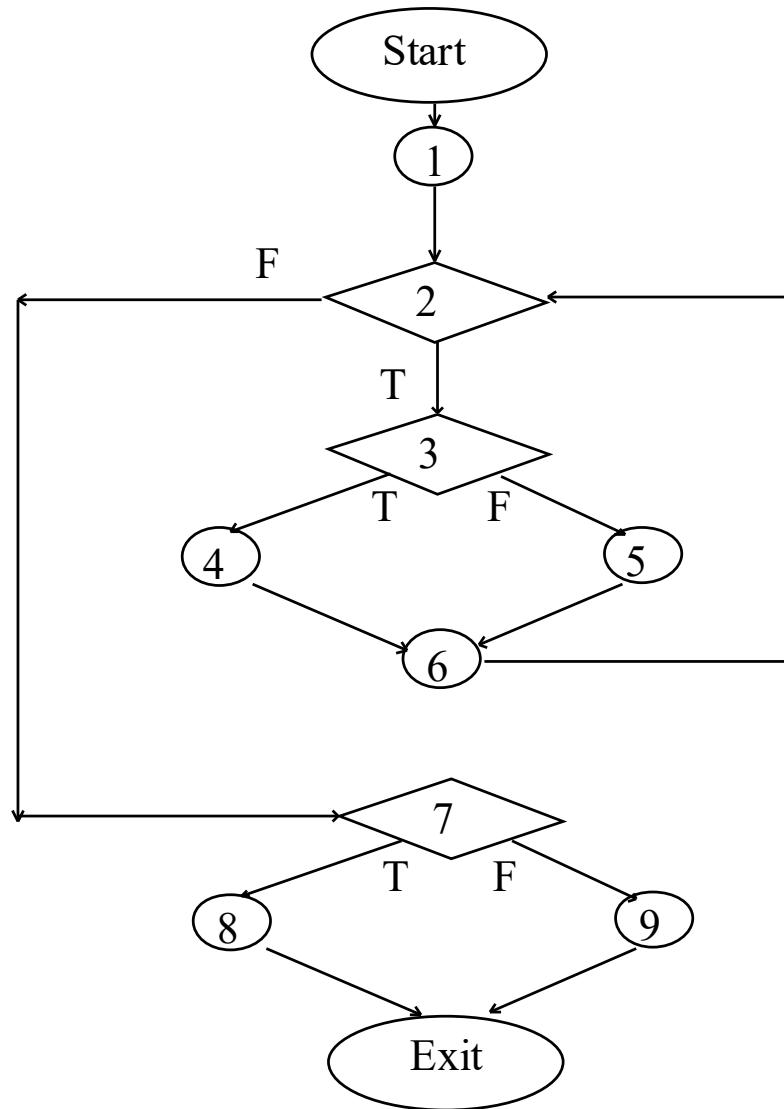
4

6

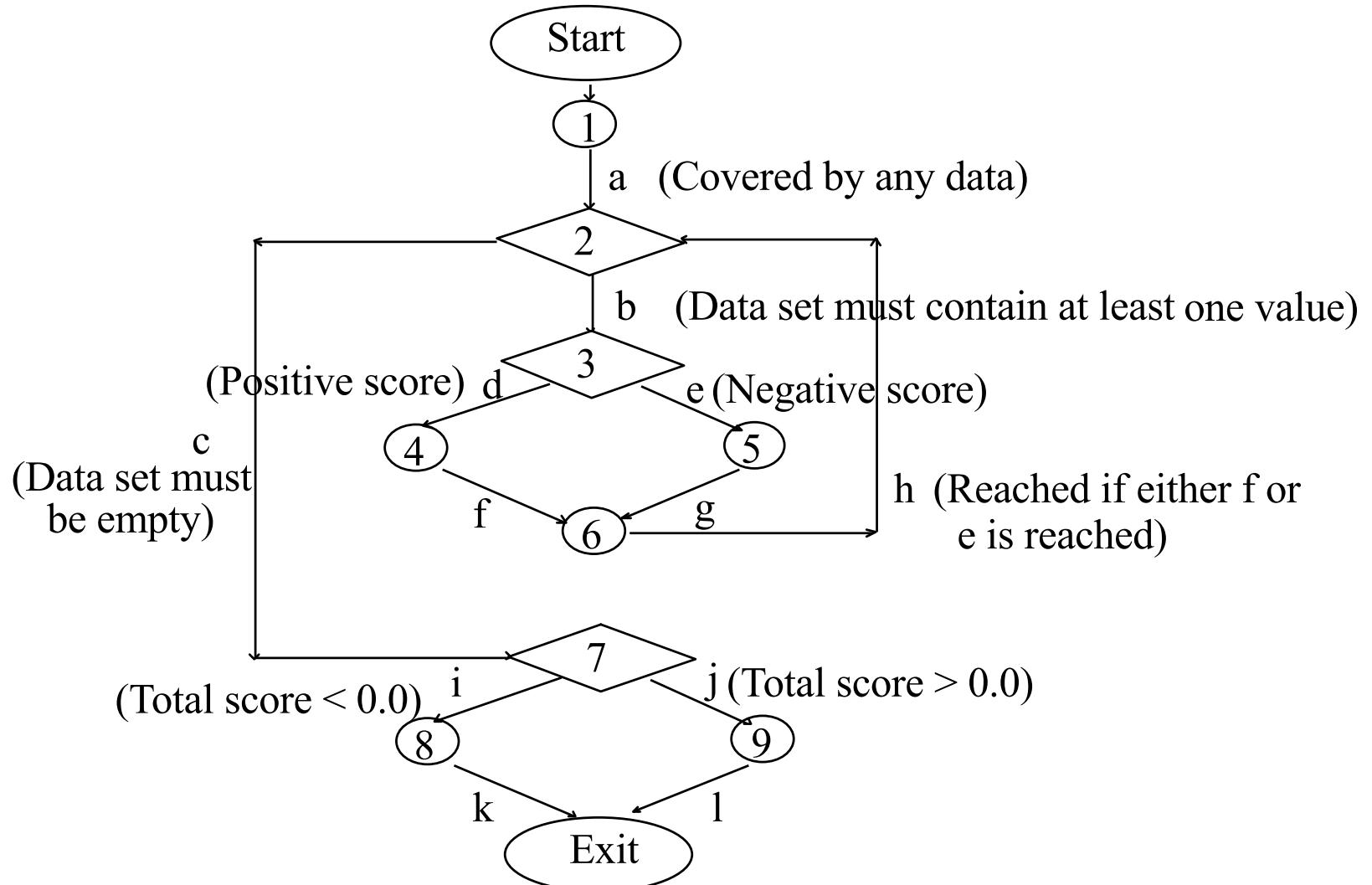
8

9

Constructing the Logic Flow Graph



Finding the Test Cases



Test Cases

- ◆ Test case 1 : ? (To execute loop exactly once)
- ◆ Test case 2 : ? (To skip loop body)
- ◆ Test case 3: ?,? (to execute loop more than once)

These 3 test cases cover all control flow paths

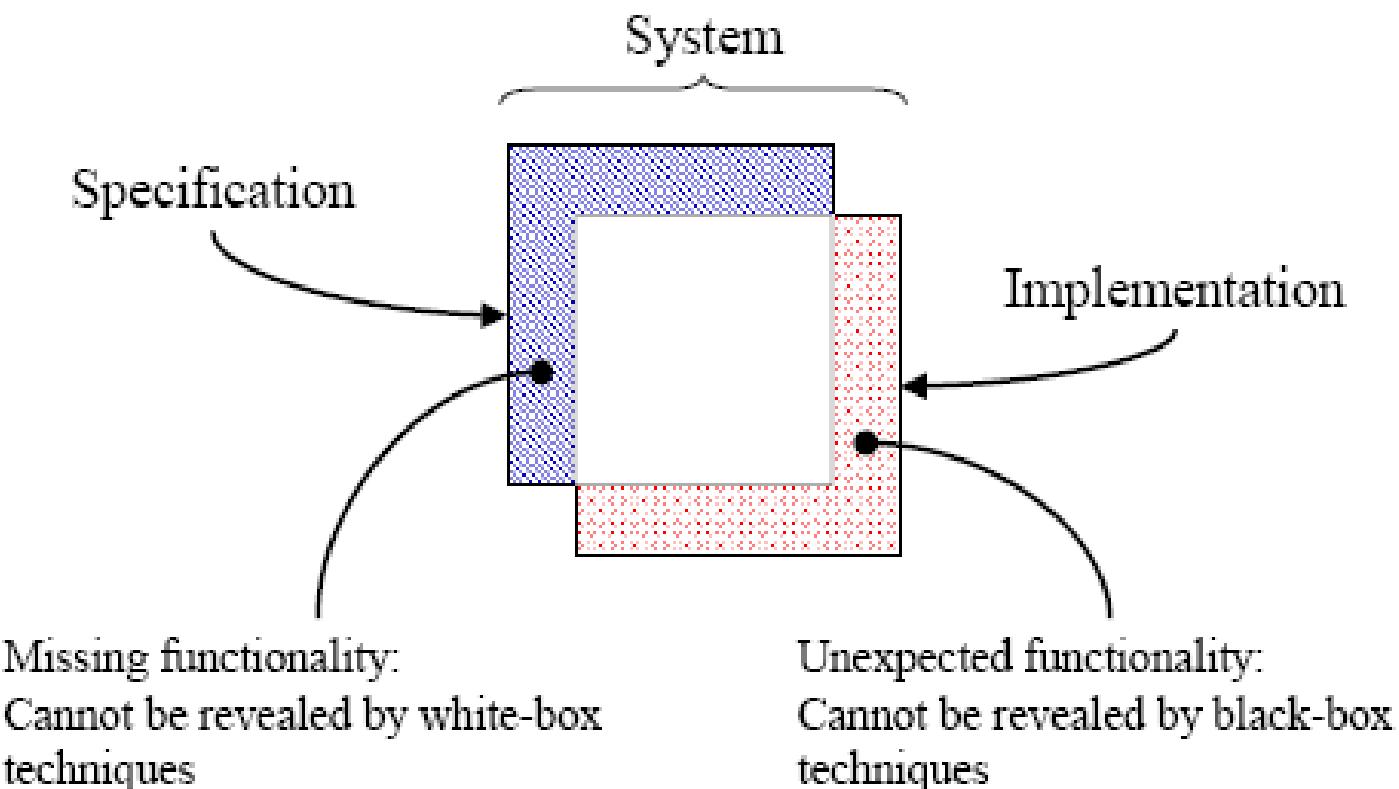
Complessità ciclomatica di McCabe: #di cammini linearmente indipendenti

Vedere esempio Libro

Comparison of White & Black-box Testing

- ◆ White-box Testing:
 - ◆ **Potentially infinite number of paths have to be tested**
 - ◆ **White-box testing often tests what is done, instead of what should be done**
 - ◆ **Cannot detect missing use cases**
- ◆ Black-box Testing:
 - ◆ **Potential combinatorical explosion of test cases (valid & invalid data)**
 - ◆ **Often not clear whether the selected test cases uncover a particular error**
 - ◆ **Does not discover extraneous use cases ("features")**
- ◆ Both types of testing are needed
- ◆ White-box testing and black box testing are the extreme ends of a testing continuum.
- ◆ Any choice of test case lies in between and depends on the following:
 - ◆ **Number of possible logical paths**
 - ◆ **Nature of input data**
 - ◆ **Amount of computation**
 - ◆ **Complexity of algorithms and data structures**

Black vs. White Box Testing



The 4 Testing Steps

1. Select what has to be measured

- ◆ **Completeness of requirements**
- ◆ **Code tested for reliability**
- ◆ **Design tested for cohesion**

2. Decide how the testing is done

- ◆ **Code inspection**
- ◆ **Black-box, white box,**
- ◆ **Select integration testing strategy (big bang, bottom up, top down, sandwich)**

3. Develop test cases

- ◆ **A test case is a set of test data or situations that will be used to exercise the unit (code, module, system) being tested or about the attribute being measured**

4. Create the test oracle

- ◆ **An oracle contains of the predicted results for a set of test cases**
- ◆ **The test oracle has to be written down before the actual testing takes place**

Guidance for Test Case Selection

- ◆ Use analysis knowledge about functional requirements (black-box):
 - ◆ Use cases
 - ◆ Expected input data
 - ◆ Invalid input data
- ◆ Use design knowledge about system structure, algorithms, data structures (white-box):
 - ◆ Control structures
 - ◆ Test branches, loops, ...
 - ◆ Data structures
 - ◆ Test records fields, arrays,
...
- ◆ Use implementation knowledge about algorithms:
 - ◆ Force division by zero

1. Create unit tests as soon as object design is completed:
 - ♦ **Black-box test: Test the use cases & functional model**
 - ♦ **White-box test: Test the dynamic model**
 - ♦ **Data-structure test: Test the object model**
2. Develop the test cases
 - ♦ **Goal: Find the minimal number of test cases to cover as many paths as possible**
3. Cross-check the test cases to eliminate duplicates
 - ♦ **Don't waste your time!**
4. Desk check your source code
 - ♦ **Reduces testing time**
5. Create a test harness
 - ♦ **Test drivers and test stubs are needed for integration testing**
6. Describe the test oracle
 - ♦ **Often the result of the first successfully executed test**
7. Execute the test cases
 - ♦ **Don't forget regression testing**
 - ♦ **Re-execute test cases every time a change is made.**
8. Compare the results of the test with the test oracle
 - ♦ **Automate as much as possible**

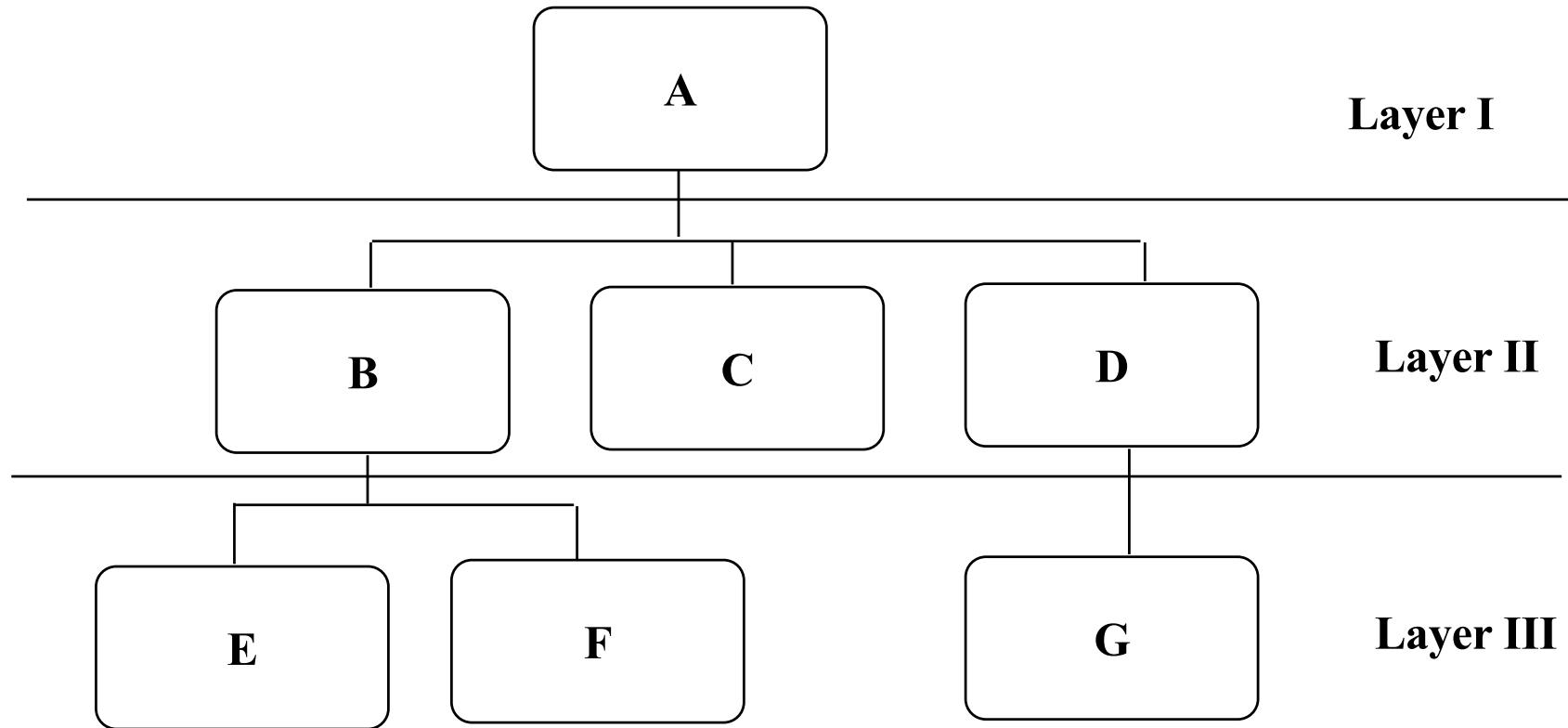
Integration testing

- ◆ Quando i bug in ogni componente sono stati rilevati e riparati, le componenti sono pronte per essere integrate in sottosistemi più grandi.
- ◆ **Integration testing** rileva bug che non sono stati determinati durante lo unit testing, focalizzandosi su un insieme di componenti che sono integrate
- ◆ Due o più componenti sono integrate e analizzate, e quando dei bug sono rilevati nuove componenti possono essere aggiunte per riparare i bug
- ◆ Sviluppare test stub e test driver per un test di integrazione sistematico è time-consuming
- ◆ L'ordine in cui le componenti sono integrate può influenzare lo sforzo richiesto per l'integrazione
 - ◆ **Vedremo diverse strategie di testing per ordinare le componenti da testare**

Component-Based Testing Strategy

- ♦ L'ordine in cui i sottosistemi sono selezionati per il testing e l'integrazione determina la strategia di testing
 - ◆ **Big bang integration (Nonincremental)**
 - ◆ **Bottom up integration**
 - ◆ **Top down integration**
 - ◆ **Sandwich testing**
 - ◆ **Variations of the above**
- ♦ Ognuna di queste strategie è stata originariamente concepita per una decomposizione gerarchica del sistema
 - ◆ **ogni componente appartiene ad una gerarchia di layer, ordinati in base all' associazione “Call”**

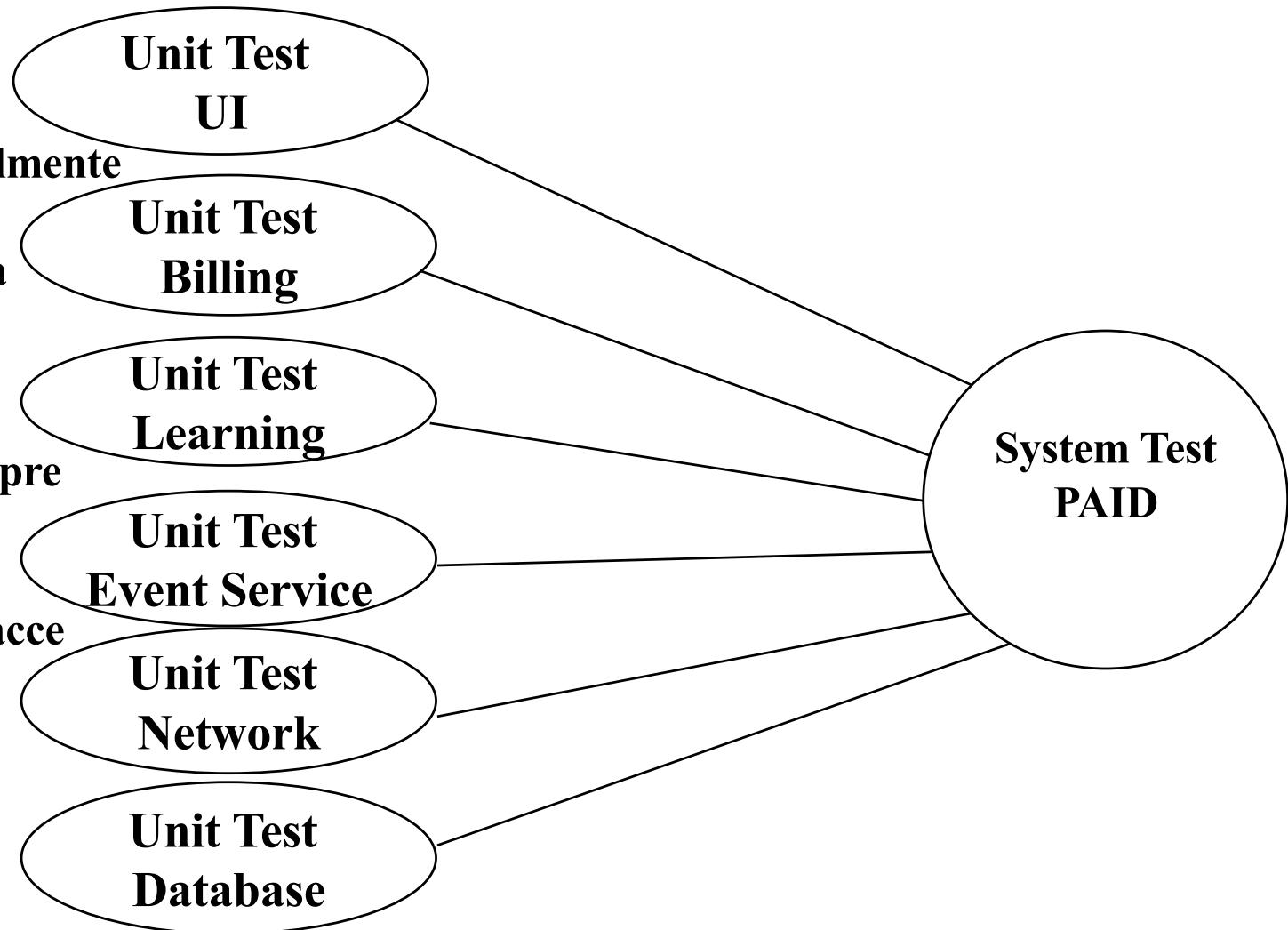
Example: Three Layer Call Hierarchy



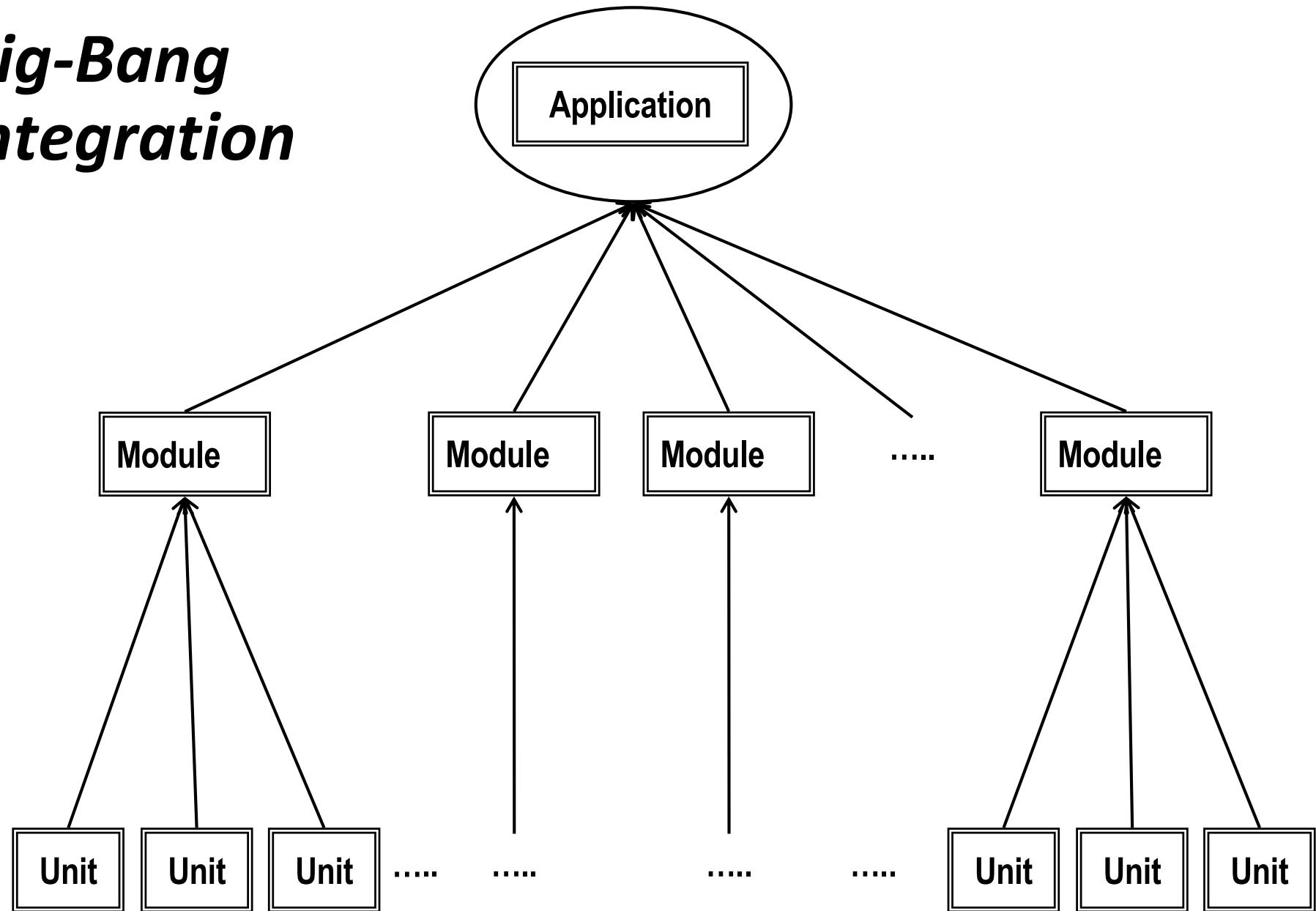
Integration Testing: Big-Bang Approach

Le componenti sono prima testate individualmente e poi testate insieme come un singolo sistema

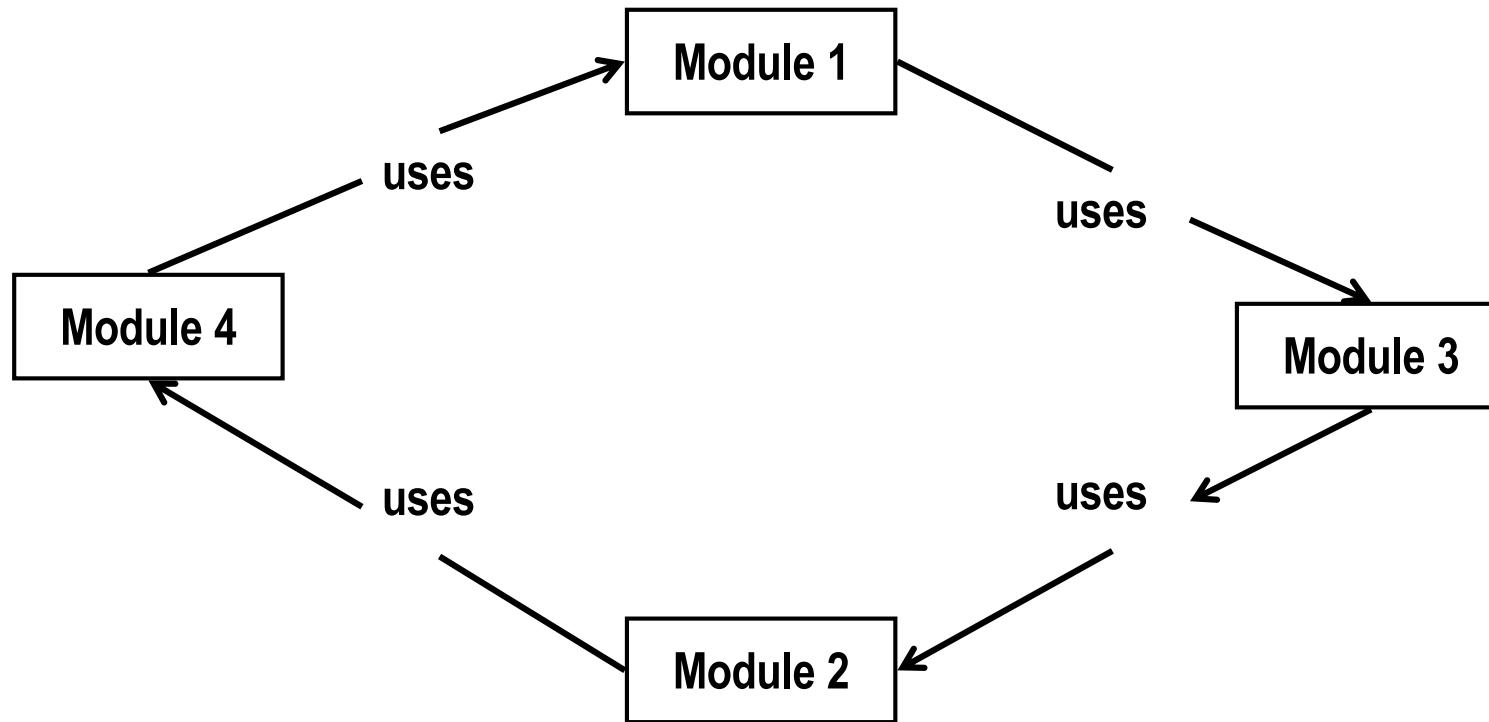
Sebbene sia semplice, è costoso: se un test scopre un fallimento è impossibile distinguere i fallimenti nelle interfacce dai fallimenti all'interno delle componenti



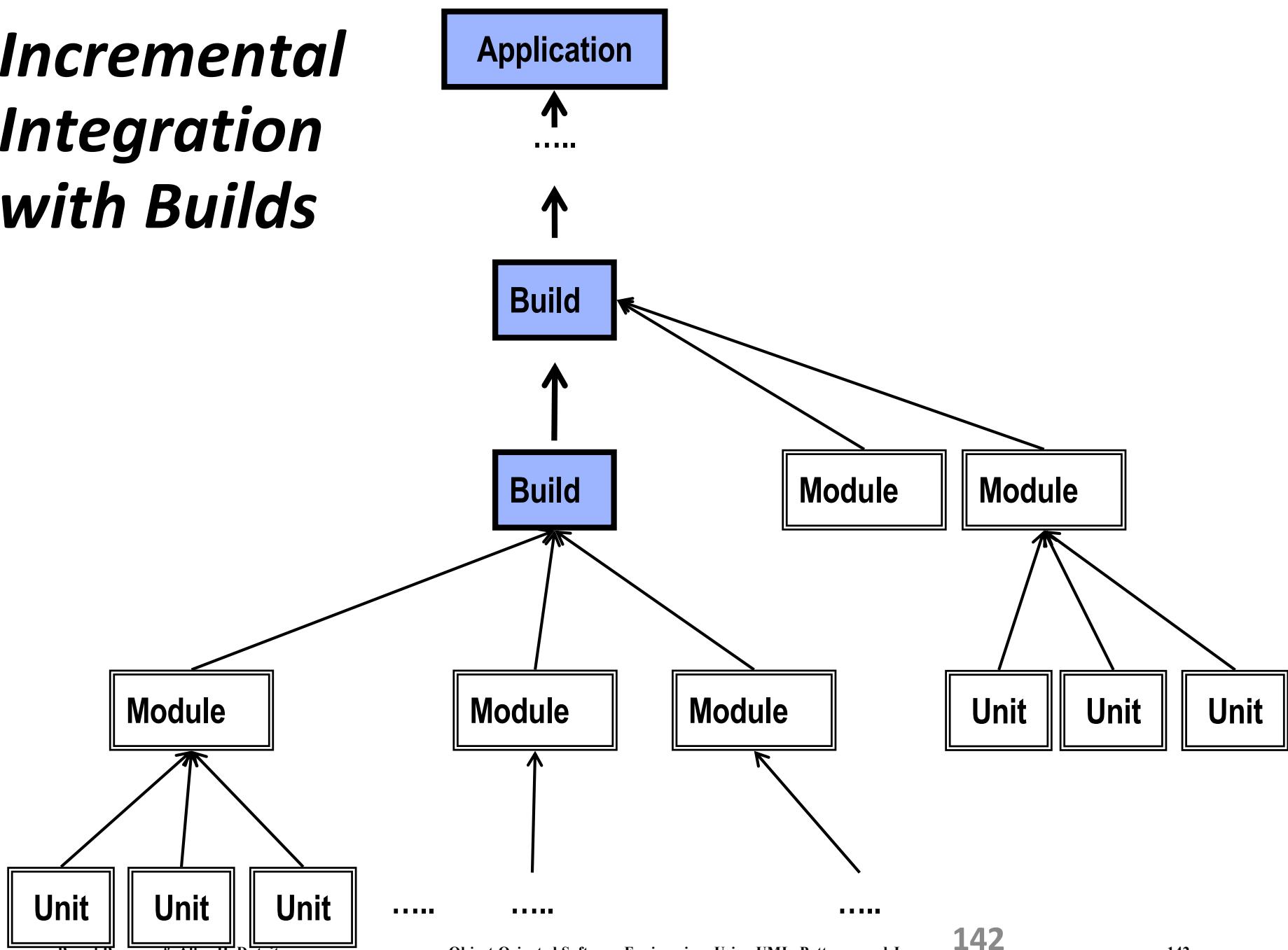
Big-Bang Integration



Module Self-Dependency → Big Bang Integration



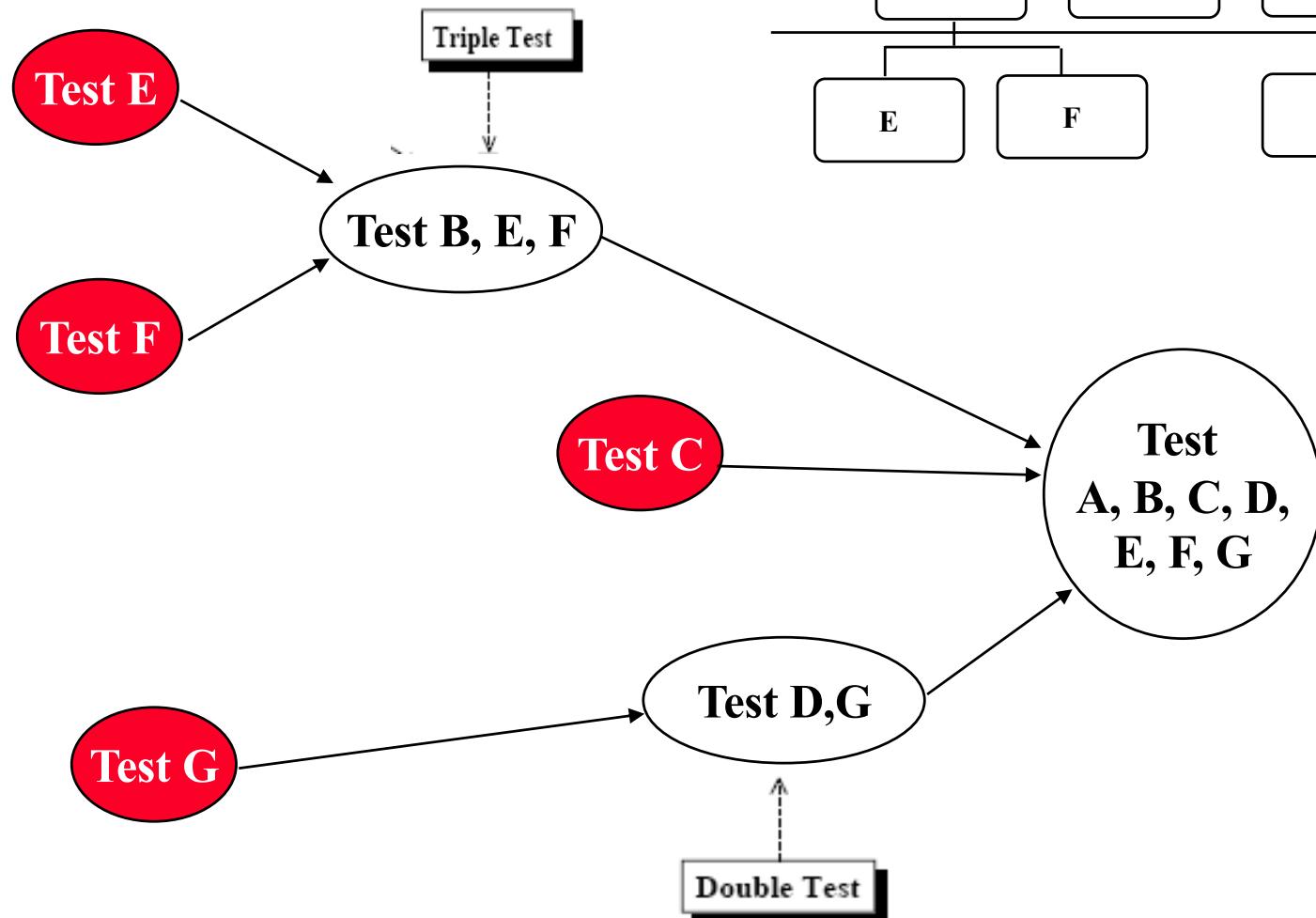
Incremental Integration with Builds



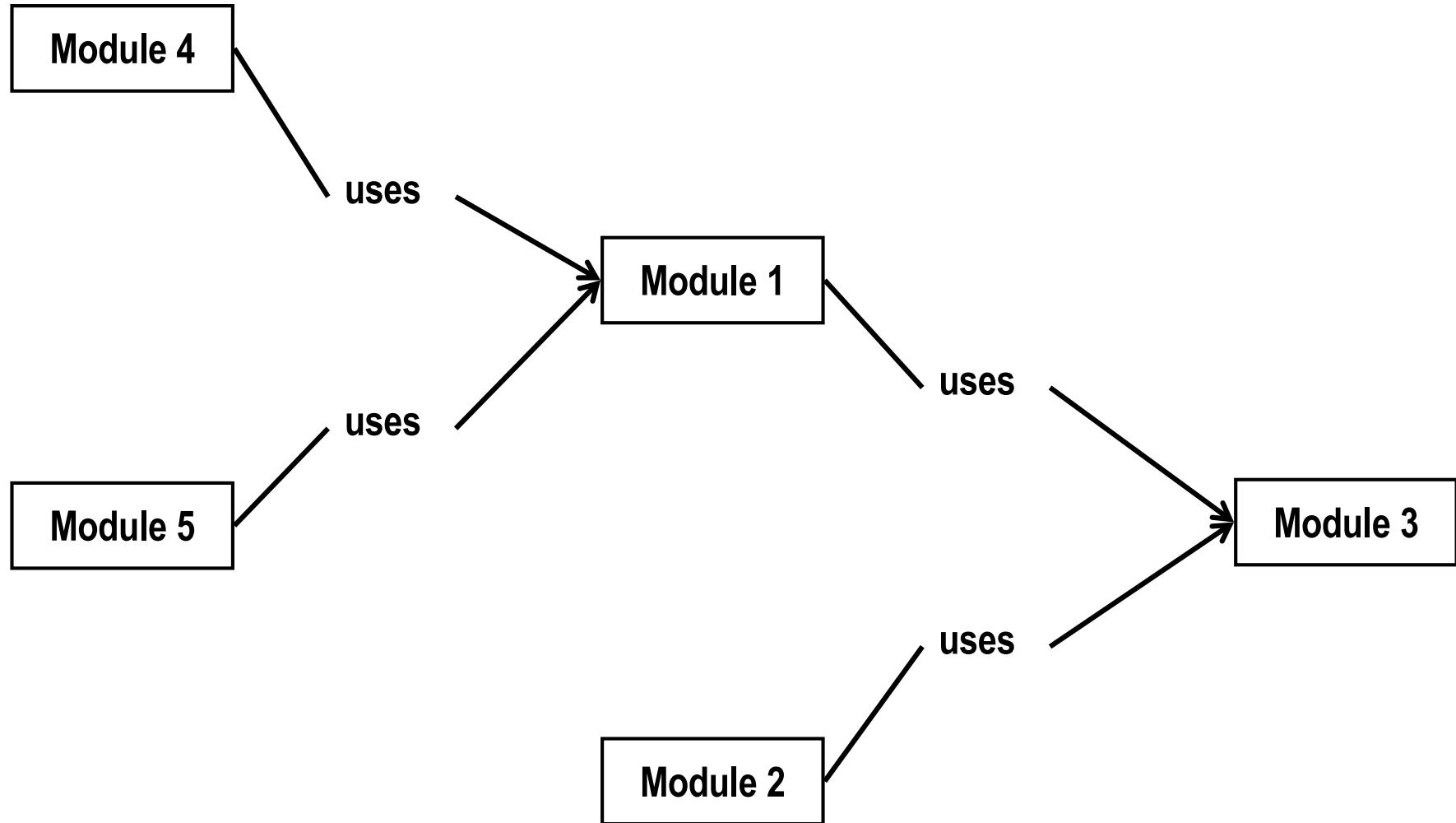
Bottom-up Testing Strategy

- ◆ I sottosistemi nel layer più in basso della gerarchia sono testati individualmente
- ◆ Poi i prossimi sottosistemi che sono testati sono quelli che “chiamano” i sottosistemi testati in precedenza
- ◆ Si ripete questo passo finché tutti i sottosistemi sono testati
- ◆ I **test driver** sono usati per simulare le componenti dei layer più “in alto” che non sono stati ancora integrati

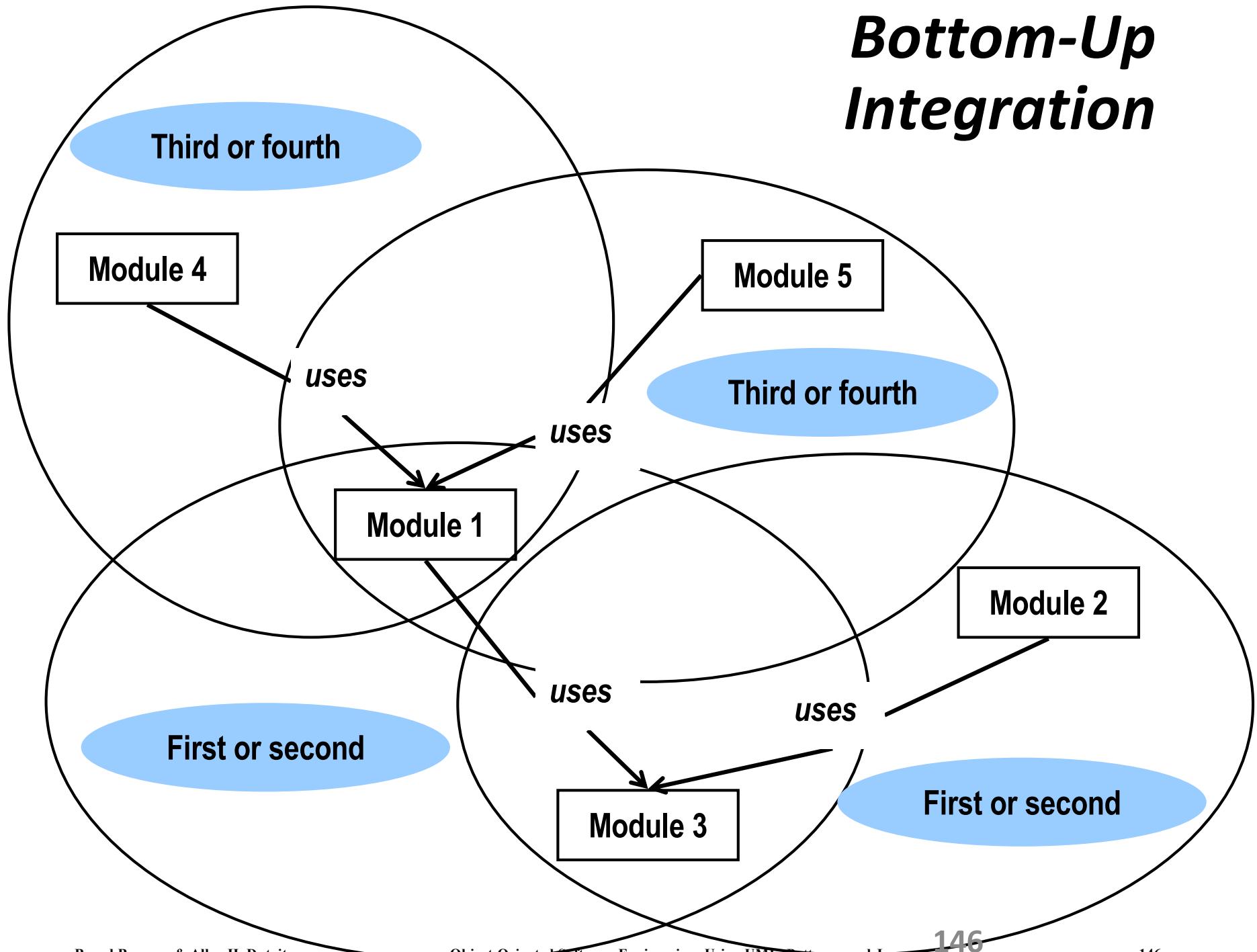
Bottom-up Integration



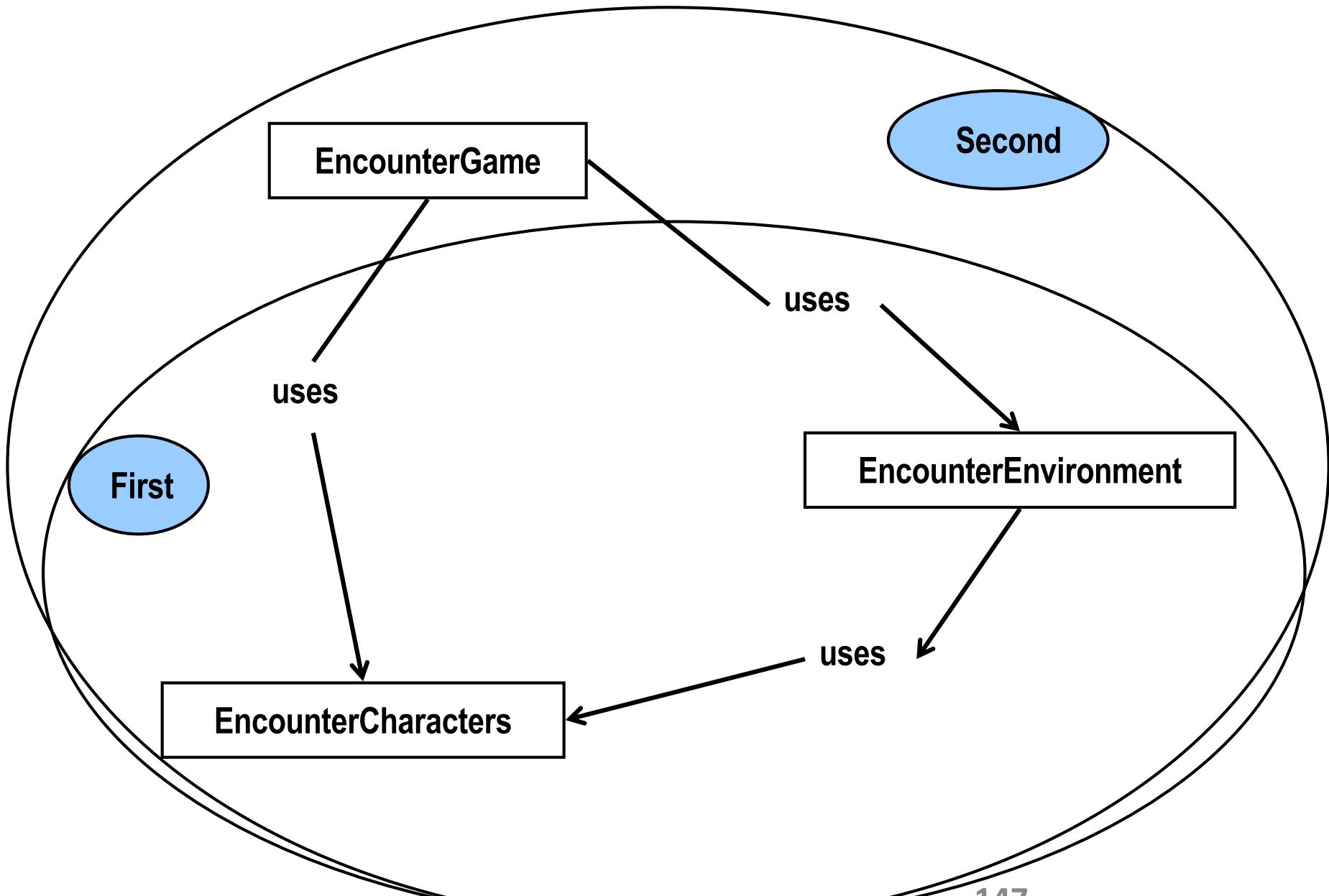
Module Dependencies



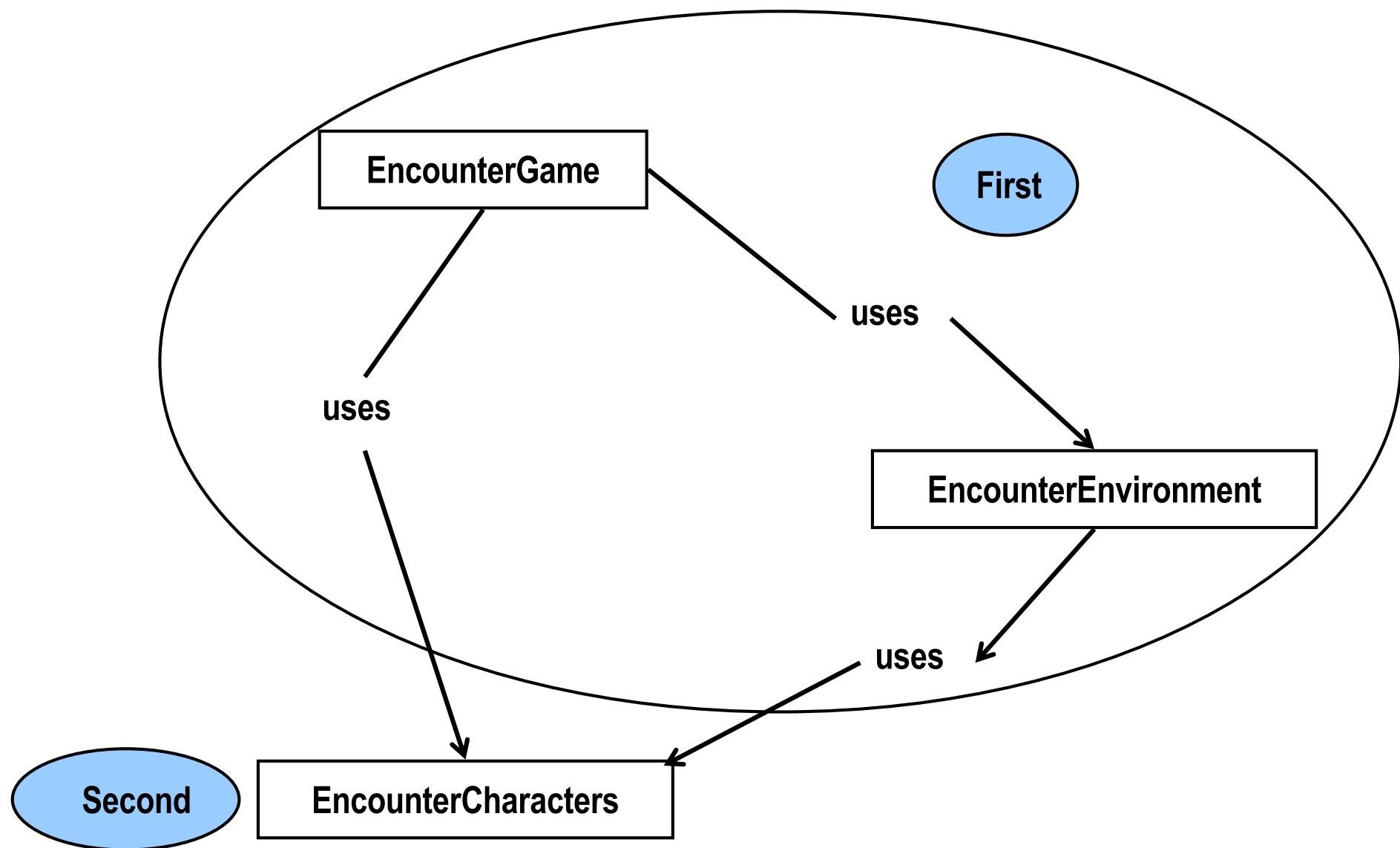
Bottom-Up Integration



Bottom-Up Integration Testing in Encounter



Top-Down Integration Testing in Encounter



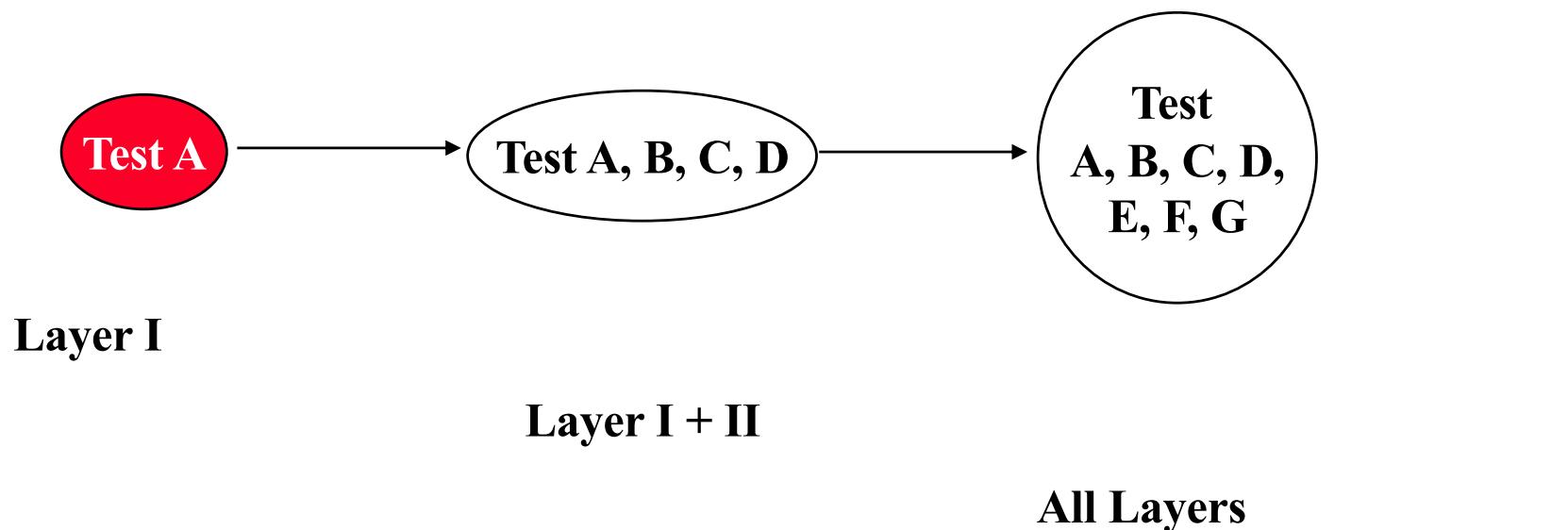
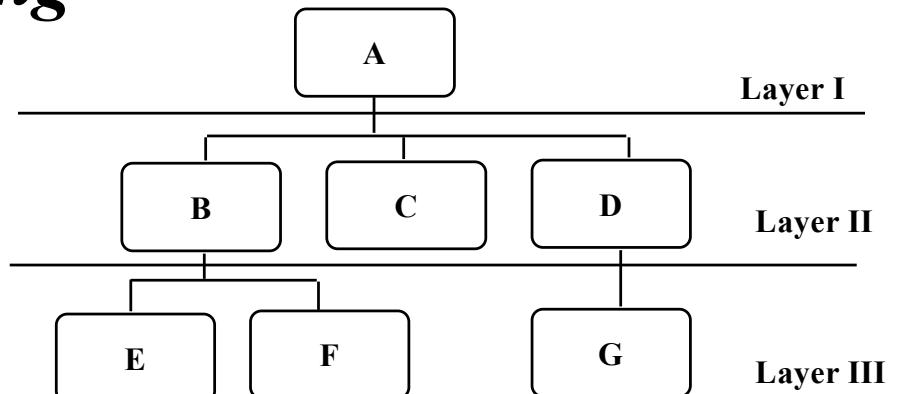
Pros and Cons of bottom up integration testing

- ◆ Non buono per sistemi decomposti funzionalmente:
 - ◆ **Testa i sottosistemi più importanti alla fine**
- ◆ Utile per integrare i seguenti sistemi
 - ◆ **Sistemi Object-oriented**
 - ◆ **Sistemi real-time**
 - ◆ **Sistemi con rigide richieste sulle performance**

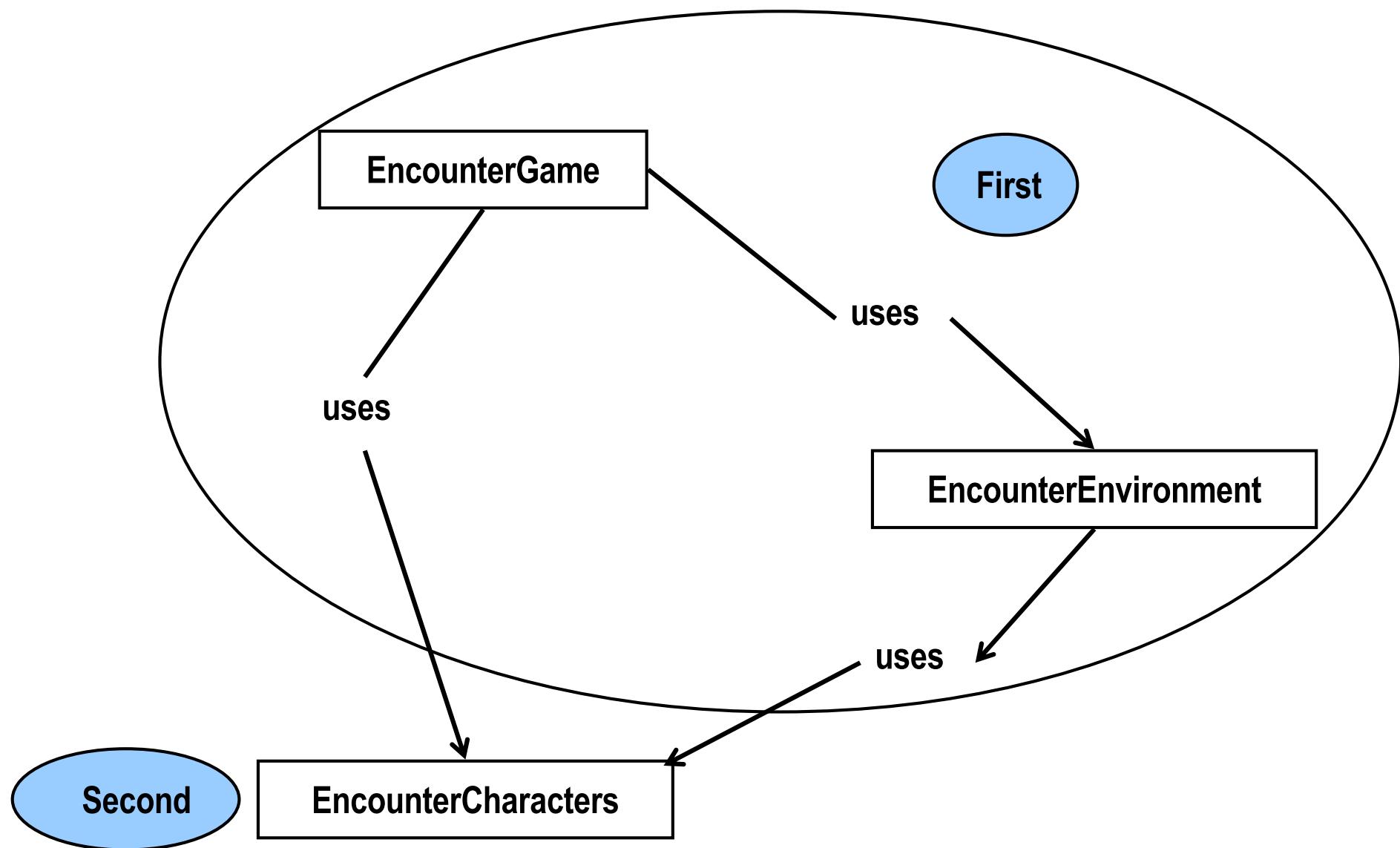
Top-down Testing Strategy

- ◆ Testa prima i layer al top o i sottosistemi di controllo
- ◆ Poi combina tutti i sottosistemi che sono chiamati dai sottosistemi testati e quindi testa la collezione risultante di sottosistemi
- ◆ Itera questa attività fino a quando tutti i sistemi non sono integrati e testati
- ◆ I **test stub** sono usati per simulare le componenti dei layer al bottom che non sono state ancora integrate
 - ◆ A program or a method that simulates the activity of a missing subsystem by answering to the calling sequence of the calling subsystem and returning back fake data.

Top-down Integration Testing



Top-Down Integration Testing in Encounter

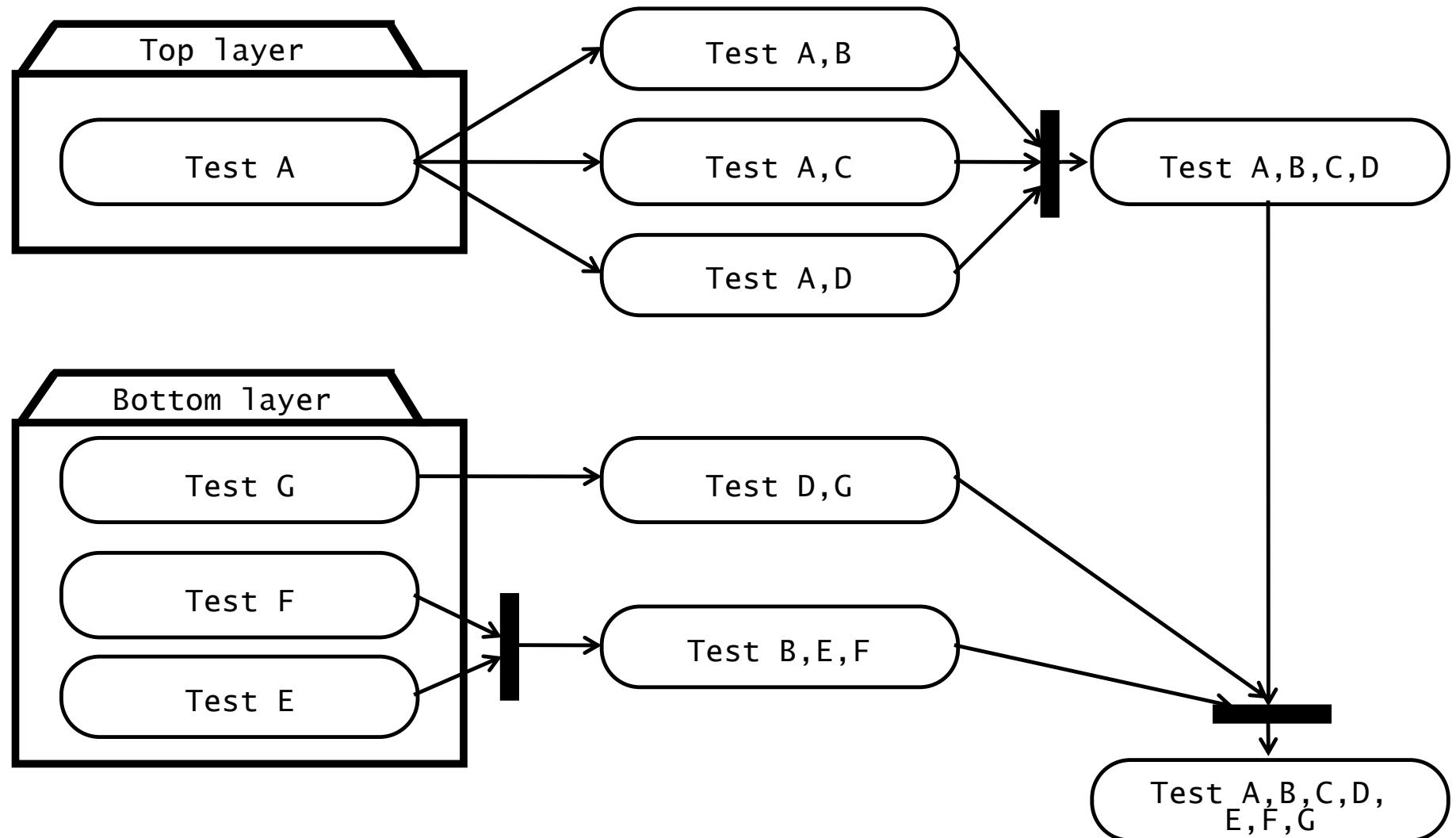


Pros and Cons of top-down integration testing

- ◆ Pros:
 - ◆ I test cases possono essere definiti in termine delle funzionalità del sistema
 - ◆ Si possono riutilizzare nelle varie iterazioni
- ◆ Cons:
 - ◆ Scrivere gli stub può essere difficile: gli stub devono consentire tutte le possibili condizioni da testare
 - ◆ È possibile che un grande numero di stub sia richiesto, specialmente se il livello più in basso del sistema contiene molti metodi.
- ◆ Una soluzione per evitare molti stub: *Modified top-down testing strategy*
 - ◆ Testa individualmente ogni layer della decomposizione prima di fare il merge dei layer
 - ◆ Svantaggio di questa modifica: Sono necessari sia test stub che test driver

Sandwich Testing Strategy

- ◆ Combina l' uso di top-down strategy con bottom-up strategy
- ◆ *Il sistema è visto come se avesse 3 layers (va riformulato)*
 - ◆ **Un layer target nel mezzo**
 - ◆ **Un layer sopra il target**
 - ◆ **Un layer sotto il target**
 - ◆ **Il testing converge al layer target**
- ◆ Come selezionare il layer target se ci sono più di 3 layers?
 - ◆ **Heuristic: tenta di minimizzare il numero di stub e di driver**



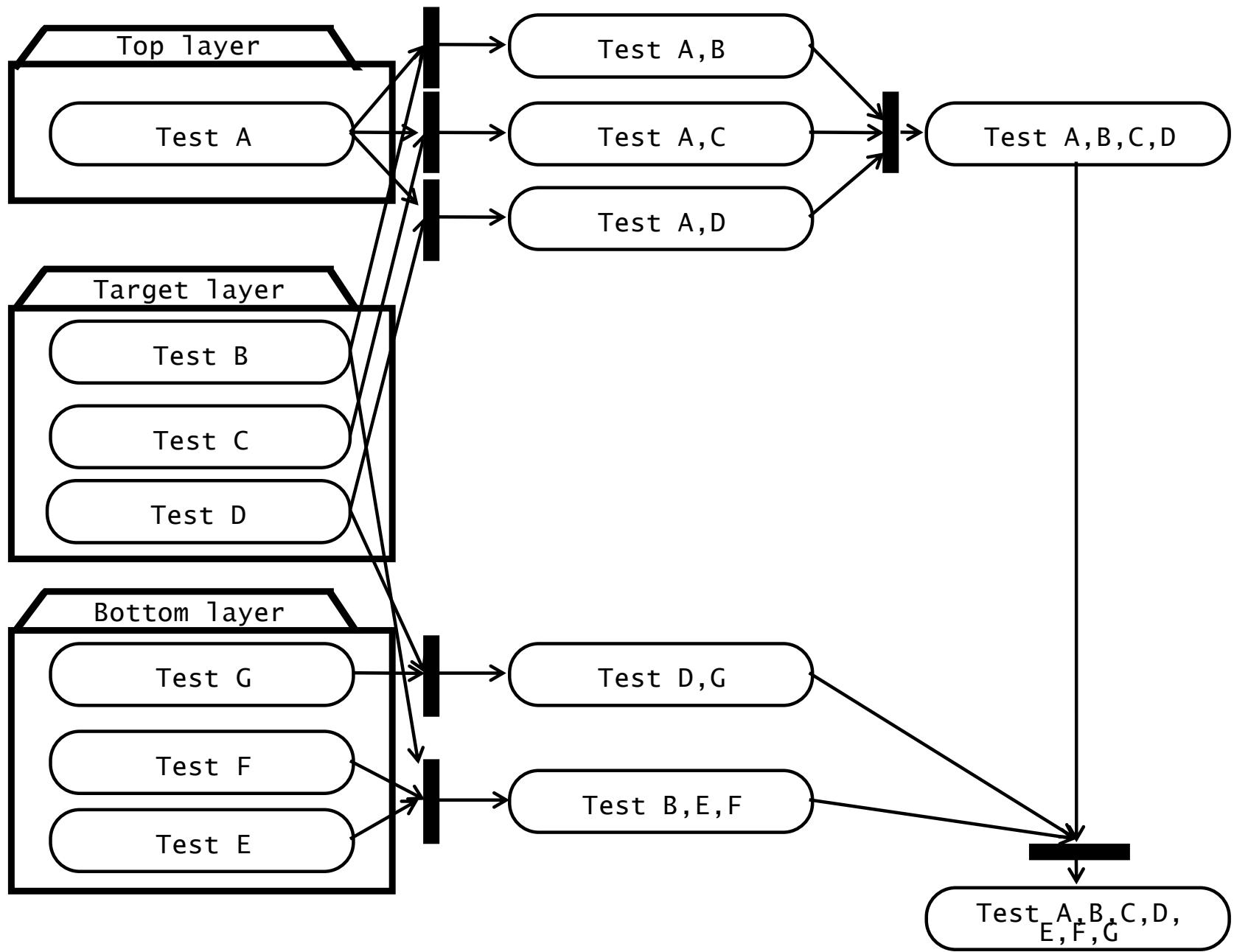
Sandwich testing strategy

Pros and Cons of Sandwich Testing

- ◆ PRO: I test dei layer al top e al bottom possono essere fatti in parallelo
- ◆ Problema: non testa i sottosistemi del target layer individualmente prima dell' integrazione
- ◆ Soluzione: **Modified sandwich testing strategy**

Modified Sandwich Testing Strategy

- ◆ Testa i 3 layer individualmente prima di combinarli in test incrementali con gli altri.
- ◆ I test individuali dei layer, consistono di un gruppo di 3 test (in parallelo):
 - ◆ **Test del layer al top (con stub per il layer target)**
 - ◆ **Test del layer nel mezzo (con driver and stub per i layer al top e al bottom rispettivamente)**
 - ◆ **Test del layer al bottom (con driver per il layer target)**
- ◆ I test dei layer combinati consistono in 2 test:
 - ◆ **Il layer al top accede al layer target. Può riusare i test del layer target dai test individuali, sostituendo i driver con le componenti del layer al top**
 - ◆ **Il layer al bottom è acceduto dal layer target. Può riusare i test del layer target dai test individuali, sostituendo gli stub con le componenti del layer al bottom**



An example of modified sandwich testing strategy.

Modified Sandwich Testing Strategy

- ◆ Test in parallel:
 - Middle layer with drivers and stubs
 - Top layer with stubs
 - Bottom layer with drivers
- ◆ Test in parallel:
 - Top layer accessing middle layer (top layer replaces drivers)
 - Bottom accessed by middle layer (bottom layer replaces stubs)

Steps in Integration Testing

1. Based on the integration strategy, *select a component* to be tested.
Unit test all the classes in the component.
2. Put selected component together; do any *preliminary fix-up* necessary to make the integration test operational (drivers, stubs)
3. Do *functional testing*: Define test cases that exercise all uses cases with the selected component
4. Do *structural testing*: Define test cases that exercise the selected component
5. Execute *performance tests*
6. *Keep records* of the test cases and testing activities.
7. Repeat steps 1 to 7 until the full system is tested.

The primary *goal of integration testing* is to identify errors in the (current) component configuration.

System Testing

- ◆ Unit testing e Integration testing si focalizzano sulla ricerca di bug nelle componenti individuali e nelle intefacce tra le componenti.
- ◆ Il System testing assicura che il sistema completo è conforme ai requisiti funzionali e non funzionali.
- ◆ Attività:
 - ◆ **Functional Testing.**
 - ◆ **Pilot Testing**
 - ◆ **Performance Testing.**
 - ◆ **Acceptance Testing**
 - ◆ **Installation Testing**
- ◆ Impatto dei requisiti sul testing del sistema:
 - ◆ Più esplicativi sono i requisiti, più facile sono da testare.
 - ◆ La qualità degli use case influenza il Functional Testing
 - ◆ La qualità della decomposizione dei sottosistemi influenza lo Structure Testing
 - ◆ La qualità dei requisiti non funzionali e dei vincoli influenza il Performance Testing

Functional Testing

Essenzialmente il black box testing

- ◆ Goal: testare le funzionalità del sistema
- ◆ I test case sono progettati dal documento dei requisiti e si focalizza sulle richieste e le funzioni chiave (gli use case)
- ◆ Il sistema è trattato come un black box.
- ◆ I test case per le unità possono essere riusati, ma devono essere scelti quelli rilevanti per l'utente finale e che hanno una buona probabilità di riscontrare un fallimento.

Performance Testing

- ◆ Stress Testing
 - ◆ **Stress limits of system (maximum # of users, extended operation)**
- ◆ Volume testing
 - ◆ **Test what happens if large amounts of data are handled**
- ◆ Configuration testing
 - ◆ **Test the various software and hardware configurations**
- ◆ Compatibility test
 - ◆ **Test backward compatibility with existing systems**
- ◆ Security testing
 - ◆ **Try to violate security requirements**
- ◆ Timing testing
 - ◆ **Evaluate response times and time to perform a function**
- ◆ Environmental test
 - ◆ **Test tolerances for heat, humidity, motion, portability**
- ◆ Quality testing
 - ◆ **Test reliability, maintainability & availability of the system**
- ◆ Recovery testing
 - ◆ **Tests system's response to presence of errors or loss of data.**
- ◆ Human factors testing
 - ◆ **Tests user interface with user**

Test Cases for Performance Testing

- ◆ Spingere il sistema integrato verso i suoi limiti!
- ◆ **Goal: tentare di rompere il sistema**
- ◆ Testare come il sistema si comporta quando è sovraccarico.
 - ◆ Possono essere identificati colli di bottiglia? (I primi candidati ad ad essere riprogettati nella prossima iterazione)
- ◆ Tenta ordini di esecuzioni non usuali
 - ◆ Call a receive() before send()
- ◆ Controlla le risposte del sistema a grandi volumi di dati
 - ◆ Se si è supposto che il sistema debba gestire 1000 item, provalo con 1001 item.

Pilot testing

- ◆ Il sistema è installato e usato da un insieme di utenti selezionati
- ◆ Nessuna linea guida o scenario è fornito agli utenti
- ◆ Sistemi pilota sono utili quando un sistema è costruito senza un insieme di richieste specifiche, o senza un particolare cliente in mente
- ◆ **Alpha test.** È un test pilota con utenti che esercitano il sistema **nell'ambiente di sviluppo**
- ◆ **Beta test.** Il test di accettazione è realizzato da un numero limitato di utenti **nell'ambiente di utilizzo**
 - ◆ **Il nuovo paradigma ampiamente utilizzato con la distribuzione del software tramite Internet**
 - ◆ **Offre il software a chiunque che è interessato a testarlo**
 - ◆ **Gli utenti sono carichi del testing del software!**

Acceptance testing

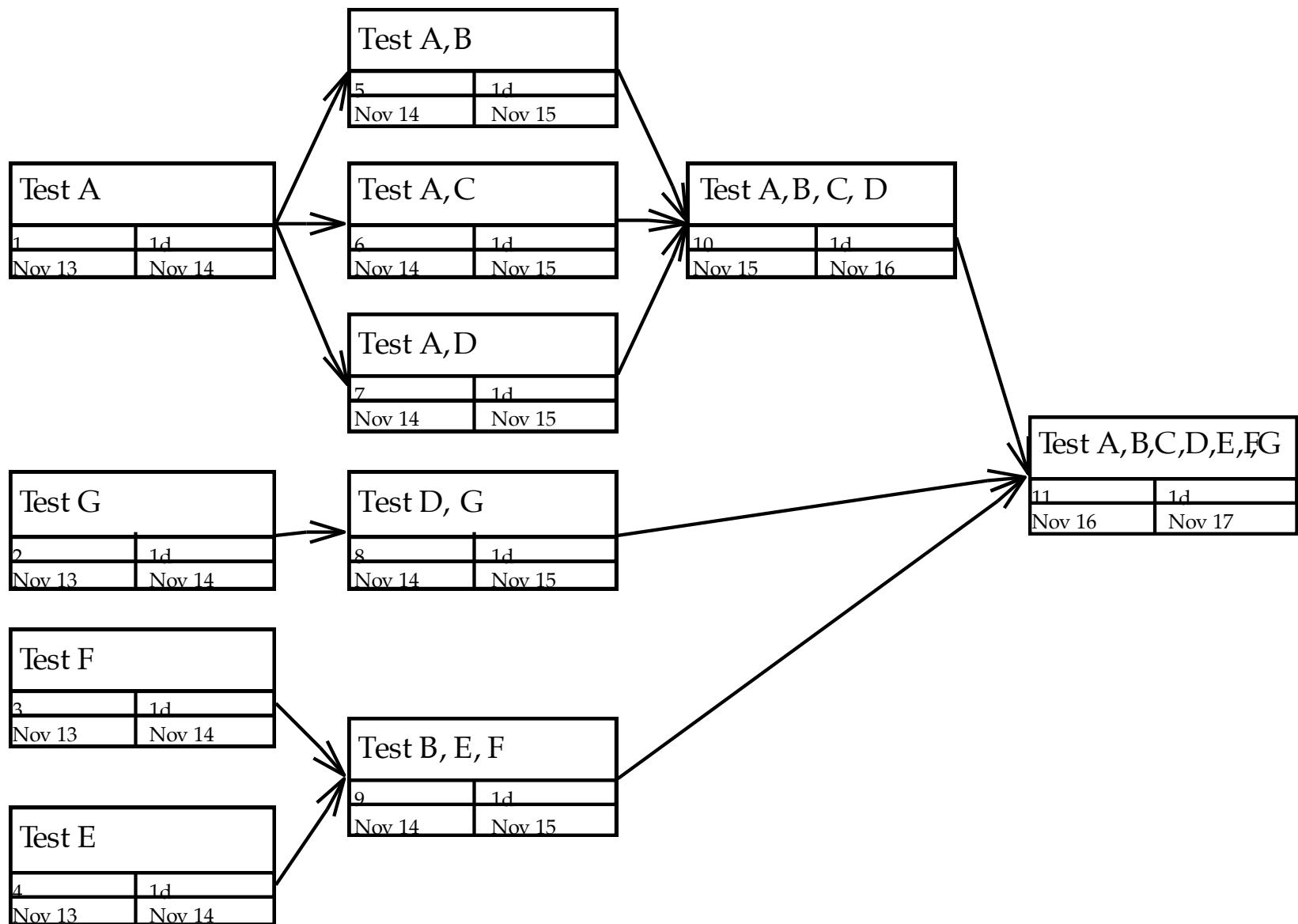
- ♦ Tre modi in cui il cliente può valutare un sistema durante l'acceptance testing:
 - ◆ **Benchmark test.** Il cliente prepara un insieme di test case che rappresentano le condizioni tipiche sotto cui il sistema dovrà operare
 - ◆ **Competitor testing.** Il nuovo sistema è testato contro un sistema esistente o un prodotto concorrente (in reengineering project)
 - ◆ **Shadow testing.** Una forma di testing a confronto, il nuovo sistema e il sistema legacy sono eseguiti in parallelo e i loro output sono confrontati
- ♦ Se il cliente è soddisfatto, il sistema è accettato, eventualmente con una lista di cambiamenti da effettuare

Installation testing

- ◆ Dopo che il sistema è accettato, esso è installato nell'ambiente di utilizzo.
- ◆ Il sistema installato deve soddisfare in pieno le richieste del cliente
- ◆ In molti casi il test di installazione ripete i test case eseguiti durante il functional testing e il performance testing nell'ambiente di utilizzo
- ◆ Quando il cliente è soddisfatto, il sistema viene formalmente rilasciato, ed è pronto per l'uso

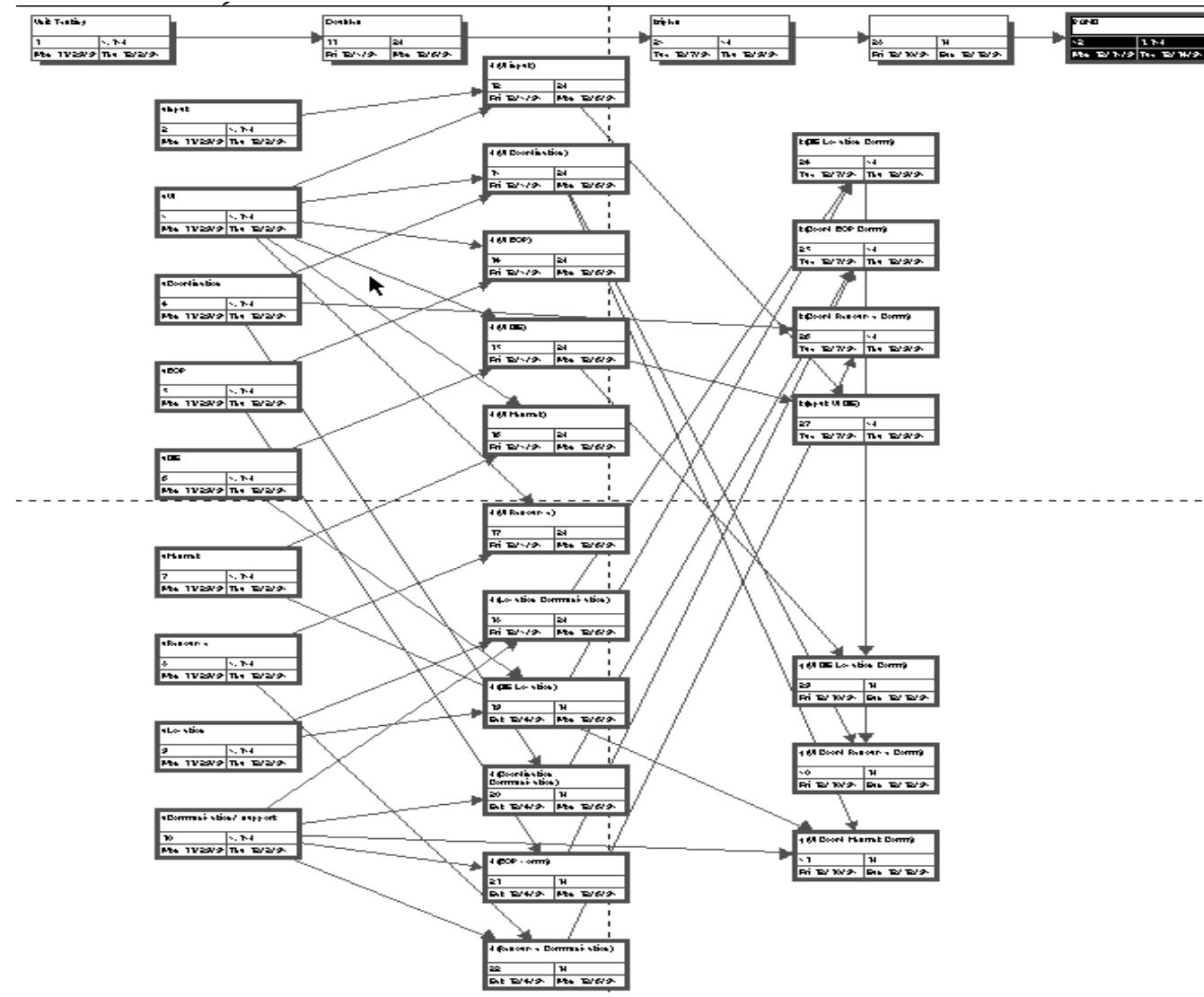
Managing testing

- ◆ Come gestire le testing activity per minimizzare le risorse necessarie
 - ◆ Alla fine, gli sviluppatori dovrebbero rilevare e quindi riparare un numero sufficiente di bug tale che il sistema soddisfi i requisiti funzionali e non funzionali entro un limite accettabile da parte del cliente
- ◆ Pianificazione delle testing activity (diagramma di PERT che mostra le dipendenze)
- ◆ Le attività sono documentate in 4 tipi di documenti:
 - ◆ il **Test Plan** che si focalizza sugli aspetti manageriali;
 - ◆ Ogni test è documentato attraverso un **Test Case Specification**;
 - ◆ Ogni esecuzione di un test case è documentata attraverso un **Test Incident Report**;
 - ◆ Il **Test Report Summary** elenca tutti i fallimenti rilevati durante i test che devono essere investigati
- ◆ Ruoli assegnati durante il testing



Example of a PERT chart for a schedule of the sandwich tests

Esempio di un diagramma di PERT (Scheduling Sandwich Tests)



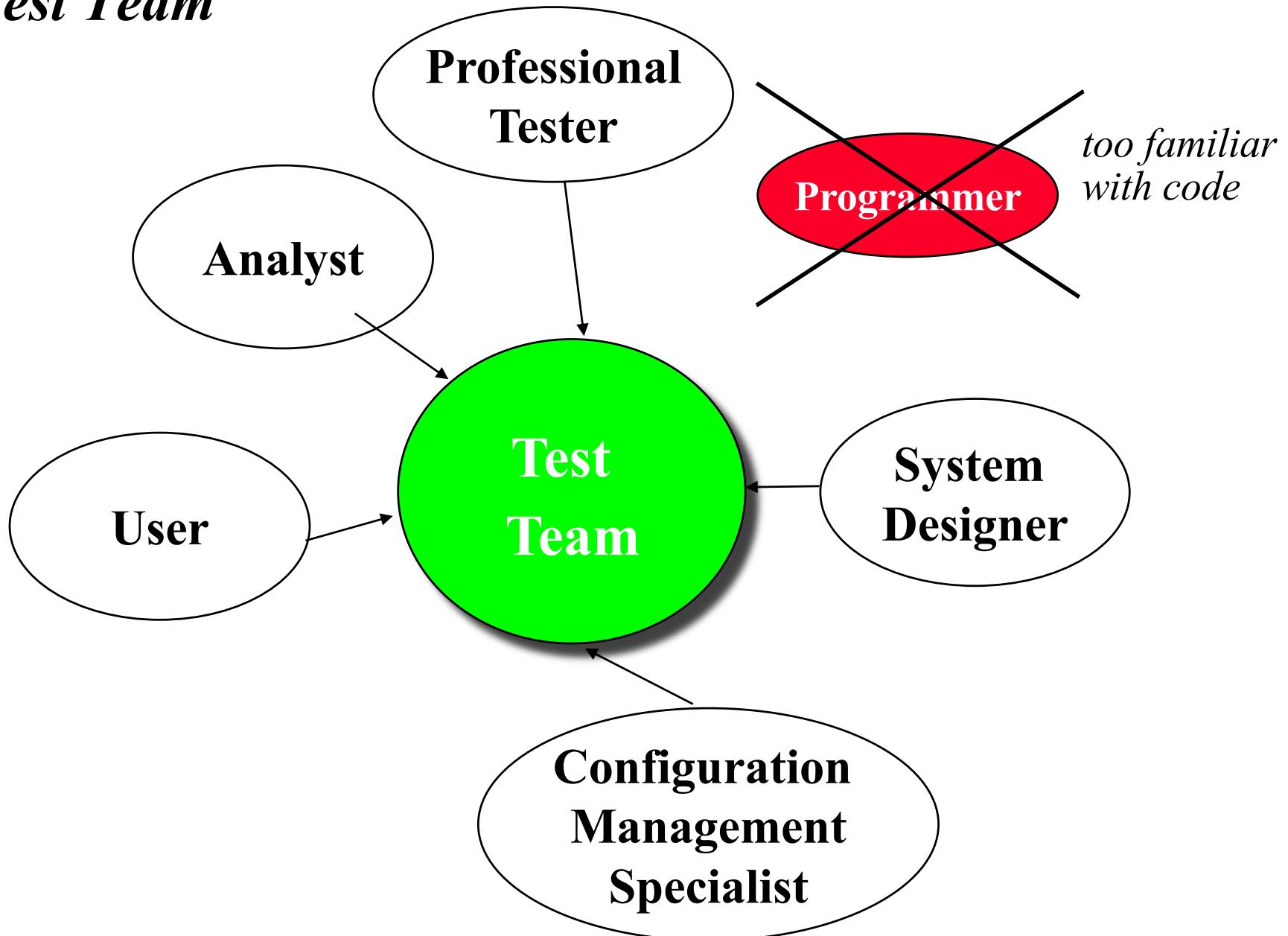
Unit Tests

Double Tests

Triple Tests

SystemTests

Test Team



Testing has its own Life Cycle

Establish the test objectives

Design the test cases

Write the test cases

Test the test cases

Execute the tests

Evaluate the test results

Change the system

Do regression testing

