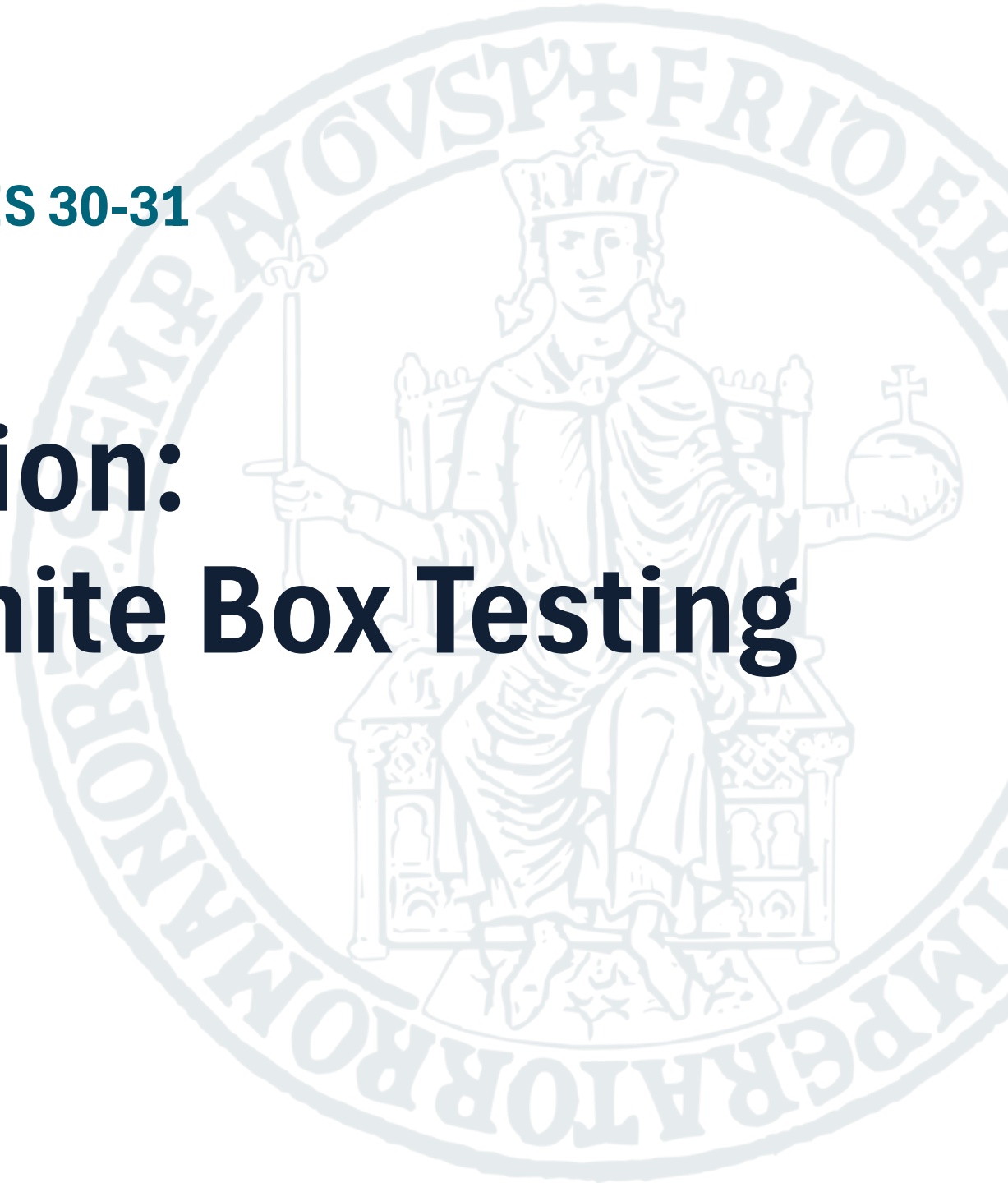
UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SOFTWARE ENGINEERING – LECTURES 30-31

Test Case Definition: Black Box and White Box Testing

Prof. Sergio Di Martino



Come testare una unità?

- Con la disponibilità di strumenti di Automated Unit Testing, la main challenge è nella definizione di un giusto “compromesso” in termini di quantità di casi di test.
- E’ impossibile testare completamente un modulo che sia “nontrivial”
  - Limitazioni teoriche: il problema della fermata
  - Limitazioni pratiche: Tempi e costi proibitivi
- “*Il Testing può solo dimostrare la presenza di bugs, non la loro assenza*” (Dijkstra)

Complete coverage?

- Supponiamo di voler testare un compilatore Java
- Input?
  - Tutti i programmi validi Java?
  - Tutti i programmi non validi Java?
  - Tutte le possibili permutazioni di stringhe?
- **Non è possibile ottenere complete coverage**.
- Servono strategie più smart per limitare il numero di casi di test

How good is our Unit Testing?

- How do we know how well they test all of our code?
- Do the unit tests execute across a broad area of code or are they simply testing the same methods and ignoring others?
- **Code Coverage** is a metric assessing how much of the code is “covered” (stressed) by a test suite.
- A few examples of code coverage measures are:
  - **Line coverage**–how many lines of code have been tested?
  - **Function coverage**– The proportion of functions or methods performed by tests.
  - **Branch coverage**– It’s a metric for determining how much of a program’s branches have been tested. A branch is a place of code where the program makes a choice, like an if statement or a loop.
  - …

Input Domains and Testing

- The input domain for a program contains all the possible inputs to that program
- For even small programs, the input domain is so large that it might as well be infinite
- **Testing is fundamentally about choosing finite sets of values from (potentially infinite) input domains**

Strategie per il testing

- Esistono due principali macro-strategie di testing:
  - **Testing Black-Box**: La definizione del casi di test è fondata sui requisiti funzionali dell'unità da testare
  - **Testing White-Box**: La definizione del casi di test è fondata sul particolari aspetti della struttura del codice dell'unità da testare

Testing Black Box

Testing Black Box

- La definizione dei casi di test è effettuata considerando esclusivamente i requisiti funzionali dell'unità da testare (la Unit è una Black Box).

- Attività principale di generazione di casi di test …
  - E' scalabile (usabile per i diversi livelli di testing)
  - Non può rilevare difetti dovuti a funzionalità extra rispetto alle specifiche
- La definizione dei casi di test prevede 5 passi

Defining Test Cases (1/5)

- Step 1 : Identify testable functions
  - Individual methods have one testable function
  - Methods in a class often have the same characteristics
  - Programs have more complicated characteristics—modeling documents such as UML can be used to design characteristics
  - Systems of integrated hardware and software components can use devices, operating systems, hardware platforms, browsers, etc.
  - Straightforward

Defining Test Cases (2/5)

- Step 2: Find all the parameters
  - Often fairly straightforward, even mechanical
  - Important to be complete
  - Methods: Parameters and state (non-local) variables used
  - Components: Parameters to methods and state variables
  - System: All inputs, including files and databases
  - Pretty straightforward

Defining Test Cases (3/5)

- Step 3 : Partition the input domain
  - The domain is scoped by the parameters
  - The structure is defined in terms of characteristics
  - Each characteristic is partitioned into sets of blocks
  - Each block represents a set of values
  - This is the most creative design step in using ISP

Defining Test Cases (4/5)

- Step 4 : Apply a test criterion to choose combinations of values
  - A test input has a value for each parameter
  - One block for each characteristic
  - Choosing all combinations is usually infeasible
  - Coverage criteria allow subsets to be chosen

Defining Test Cases (5/5)

- Step 5 : Refine combinations of blocks into test inputs
  - Choose appropriate values from each block

Step 3: Input Space Partitioning (o Classi di Equivalenza)

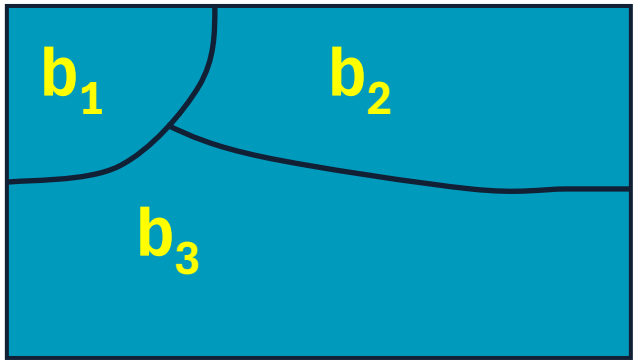
- Input parameters define the scope of the input domain
  - Parameters to a method
  - Data read from a file
  - Global variables
  - User level inputs
- ISP is a software testing strategy that divides the input parameters of a software unit into **Partitions** or **Equivalence Classes**, from which test cases can be derived.

Equivalence Partitioning

- Razionale di fondo: se un caso di test non rileva fallimenti per un valore di una classe di equivalenza, allora probabilmente non rileva fallimento per ogni altro elemento della classe.
- Non serve scrivere un caso di test per ogni valore della classe ma solo per uno o alcuni suoi rappresentanti.
- La validità dell'assunto dipende dalla ragionevole progettazione della partizione del dominio in classi di equivalenza.

Partitioning Domains

- Domain D
- Partition scheme q of D
- The partition q defines a set of blocks, $B_q = b_1 , b_2 , \ldots, b_Q$
- The partition must satisfy two properties :
  - Blocks must be pairwise disjoint (no overlap)
  - Together the blocks cover the domain D (complete)

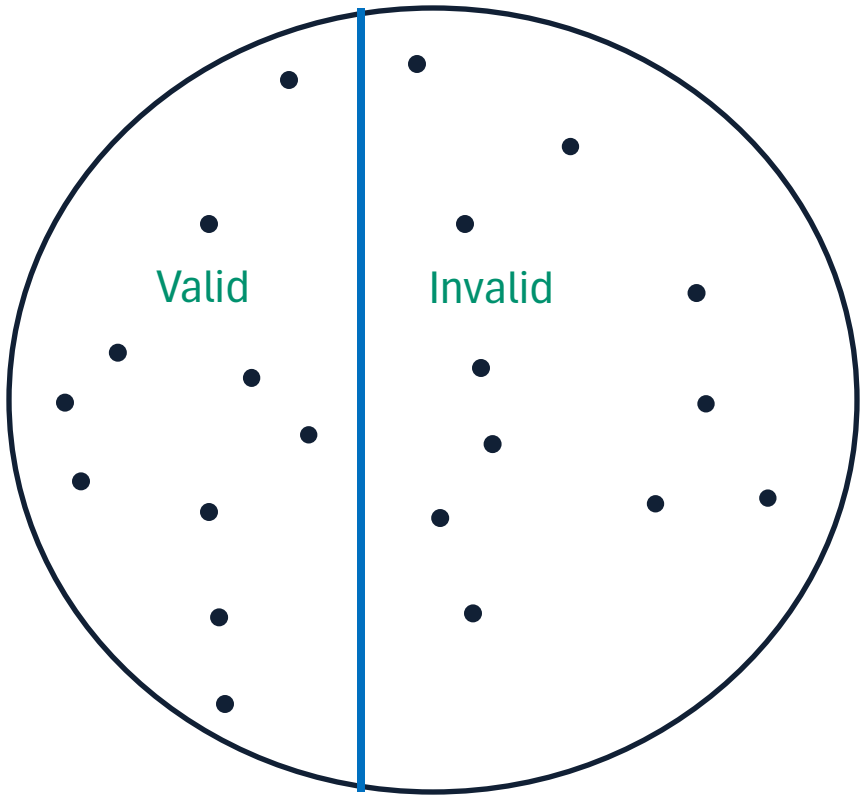


$$b_i \cap b_j = \Phi, \forall\, i \neq j,\ b_i, b_j \in B_q$$

$$\bigcup_{b \in Bq} b = D$$

Equivalence Partitioning

- Example of partitioning: Valid vs. Invalid test cases



Come identificare le Classi di Equivalenza – Enumerazione

- Se la variabile di input è un elemento di un insieme discreto (enumerazione):
  - una classe valida per ogni elemento dell'insieme, una non valida per un elemento non appartenente
  - include il caso di valori specifici
  - include il caso di valori booleani
- Es: i colori di un semaforo
  - Definiamo 4 classi di equivalenza
    - CE1 = "Rosso" (valida)
    - CE2 = "Giallo" (valida)
    - CE3 = "Verde" (valida)
    - CE4 = "Blu" (non valida)

Come identificare le Classi di Equivalenza – Intervalli di valori

- Se il parametro di input assume valori validi in un intervallo di valori di un dominio ordinato:
  - almeno una classe valida per valori interni all'intervallo, una non valida per valori inferiori al minimo, e una non valida per valori superiori al massimo
- Es: supponiamo una variabile usata per rappresentare il voto d'esame universitario.
  - La variabile ha senso nel range [0..30], con particolare riferimento al sottorange [18..30]
  - Definiamo 4 classi di equivalenza:
    - CE1 = valori ≥18 e ≤30 (esame superato)
    - CE2 = valori ≥0 e ≤17 (esame non superato)
    - CE3 = tutti i valori < 0 (non valida)
    - CE4 = tutti i valori >30 (non valida)

Two Approaches to identify Partitions

- Signature-based approach
  - Develops characteristics directly from individual input parameters
  - Simplest application
  - Can be partially automated in some situations

- Functionality-based approach
  - Develops characteristics from a behavioral view of the unit under test
  - Harder to develop—requires more design effort
  - May result in better tests, or fewer tests that are as effective

1. Signature-Based Approach

- Mechanically consider each parameter in isolation
- This is an easy modeling technique and relies mostly on syntax
- Some domain and semantic information won’t be used
  - Could lead to an incomplete IDM
- Ignores relationships among parameters

Example

```
public class TriangleType
{
   /**
     * @param s1, s2, s3:  sides of the triangle
     * @return enum describing type of triangle
     */
   public Triangle CheckTriangle (int s1, int s2, int s3)
   {
         if (s1 <= 0 || s2 <= 0 || s3 <= 0) // Reject non-positive sides
          return (Triangle.INVALID);
         if (s1+s2 <= s3 || s2+s3 <= s1 || s1+s3 <= s2) // Triangle inequality
          return (Triangle.INVALID);
         if ((s1 == s2) && (s2 == s3)) // Equilateral triangles
          return Triangle.EQUILATERAL;
         if ((s1 == s2) || (s2 == s3) || (s1 == s3)) // Isosceles triangles
          return Triangle.ISOSCELES;
      return (Triangle.SCALENE);
   }
}
```

Classi di Equivalenza per Triangolo (sol. Signature-based)

Partizioni	CE1	CE2	CE3
Relazione di S1 con 0	>0	=0	<0
Relazione di S2 con 0	>0	=0	<0
Relazione di S3 con 0	>0	=0	<0

- A maximum of 3*3*3 = 27 tests
- Some triangles are valid, some are invalid
- Refining the characterization can lead to more tests …

Functionality-Based Approach

- Identify characteristics that correspond to the intended functionality
- Requires more design effort from tester
- Can incorporate domain and semantic knowledge
- Can use relationships among parameters
- Modeling can be based on requirements, not implementation
- The same parameter may appear in multiple characteristics, so it’s harder to translate values to test cases

Classi di Equivalenza per Triangolo (sol. Functionality-based)

Partizioni	CE1	CE2	CE3	CE4
Tipo Triangolo	Scaleno	Equilatero	Isoscele	Non Valido

Progettazione dei casi di test (1)

- Le classi di equivalenza individuate devono essere utilizzate per identificare casi di test che:
  - Minimizzino il numero complessivo di test
  - Risultino significativi (affidabili)

- Euristiche di base:
  - si devono individuare tanti casi di test da coprire tutte le classi di equivalenza valide, con il vincolo che ciascun caso di test comprenda il maggior numero possibile di classi valide ancora scoperte.
  - Si devono individuare tanti casi di test da coprire tutte le classi di equivalenza non valide, con il vincolo che ciascun caso di test copra una ed una sola delle classi non valide

Funzioni con più parametri

- Come ci si comporta qualora una funzione abbia più di un parametro?
- Si procede con una combinazione delle classi di equivalenza delle varie caratteristiche seguendo opportuni criteri di copertura delle classi legate alle caratteristiche.
- La letteratura suggerisce diverse strategie di copertura con implicazioni diverse sul numero dei test generati.

Choosing Combinations of Values

- Once characteristics and partitions are defined, the next step is to choose test values
- The most obvious criterion is to choose all combinations
- Dato il prodotto cartesiano delle classi di equivalenza delle caratteristiche, per ogni elemento del prodotto cartesiano ci deve essere un caso di test che usa (simultaneamente) i valori nominali di ciascuna delle classi nell'elemento del prodotto cartesiano considerato
- All Combinations (ACoC) : All combinations of blocks from all characteristics must be used.
- Detto anche **Strong Normal Equivalence Class Testing – N-SECT**
- Number of tests is the cartesian product of the number of equivalence classes

ISP Criteria – All Combinations

- Consider the "second characterization" of Triang as given before:

Partizioni	C1	C2	C3	C4
Lunghezza Lato S1	>1	=1	=0	<0
Lunghezza Lato S2	>1	=1	=0	<0
Lunghezza Lato S3	>1	=1	=0	<0

- For convenience, we relabel the blocks using abstractions:

Characteristic	b_1	b_2	b_3	b_4
A	A1	A2	A3	A4
B	B1	B2	B3	B4
C	C1	C2	C3	C4

ISP Criteria – ACoC Tests

A1 B1 C1	A2 B1 C1	A3 B1 C1	A4 B1 C1
A1 B1 C2	A2 B1 C2	A3 B1 C2	A4 B1 C2
A1 B1 C3	A2 B1 C3	A3 B1 C3	A4 B1 C3
A1 B1 C4	A2 B1 C4	A3 B1 C4	A4 B1 C4
A1 B2 C1	A2 B2 C1	A3 B2 C1	A4 B2 C1
A1 B2 C2	A2 B2 C2	A3 B2 C2	A4 B2 C2
A1 B2 C3	A2 B2 C3	A3 B2 C3	A4 B2 C3
A1 B2 C4	A2 B2 C4	A3 B2 C4	A4 B2 C4
A1 B3 C1	A2 B3 C1	A3 B3 C1	A4 B3 C1
A1 B3 C2	A2 B3 C2	A3 B3 C2	A4 B3 C2
A1 B3 C3	A2 B3 C3	A3 B3 C3	A4 B3 C3
A1 B3 C4	A2 B3 C4	A3 B3 C4	A4 B3 C4
A1 B4 C1	A2 B4 C1	A3 B4 C1	A4 B4 C1
A1 B4 C2	A2 B4 C2	A3 B4 C2	A4 B4 C2
A1 B4 C3	A2 B4 C3	A3 B4 C3	A4 B4 C3
A1 B4 C4	A2 B4 C4	A3 B4 C4	A4 B4 C4

ACoC yields 4*4*4 = 64 tests for Triang!

This is almost certainly more than we need

Only 8 are valid (all sides greater than zero)

Weak Normal Equivalence Class Testing (N-WECT)

- 64 tests for triang() is almost certainly way too many
- One criterion comes from the idea that we should try at least one value from each equivalence class
- **Weak Normal Equivalence Class Testing – N-WECT**: Per ogni classe di equivalenza ci deve essere un Test case che usa un valore da quella classe di equivalenza.
  - Se possibile il valore nominale o il valore medio della classe
  - (Possibile per classi superiormente e inferiormente limitate e ordinate)
  - N-WECT è anche detto Each Choice Coverage (ECC)
  - Number of tests is the number of Equivalence Classes in the largest partitioning

Esempio N-WECT:
int triangolo(int a, int b, int c)

- Si osservi che non esistono test che provano per valori non ammessi i singoli parametri separatamente
- Non ci sono test che provano il requisito che la somma di due lati deve essere superiore al terzo per valori positivi dei lati

Test	a	b	c	oracolo
T1	3	3	3	0
T2	0	0	0	-1
T3	3	3	5	1
T4	3	4	5	2

Weak Robust Equivalence Class Testing (R-WECT)

- Name is admittedly counterintuitive and oxymoronic (both *weak* and *robust*?)
- It's an extension of Weak Normal Equivalence Class Testing (N-WECT)
  1. For valid classes, use one value from each valid class (an in N-WECT)
  2. For invalid classes, there should be one test with a value from the invalid class, and all values for the remaining parameters should be valid.

ISP Criteria – Pair-Wise

- Another approach combines values with other values
- Pair-Wise Coverage (PWC) : A value from each block for each characteristic must be combined with a value from every block for each other characteristic.
- Number of tests is at least the product of two largest characteristics

$$(\text{Max}_{i=1}^{Q} (B_i)) * (\text{Max}_{j=1,\, j!=i}^{Q} (B_j))$$

For *triang*() :

A1, B1, C1	A1, B2, C2	A1, B3, C3	A1, B4, C4
A2, B1, C2	A2, B2, C3	A2, B3, C4	A2, B4, C1
A3, B1, C3	A3, B2, C4	A3, B3, C1	A3, B4, C2
A4, B1, C4	A4, B2, C1	A4, B3, C2	A4, B4, C3

ISP Criteria –T-Wise

- A natural extension is to require combinations of t values instead of 2
- t-Wise Coverage (TWC) : A value from each block for each group of t characteristics must be combined.
- Number of tests is at least the product of t largest characteristics
- If all characteristics are the same size, the formula is
- If t is the number of characteristics Q, then all combinations $(\text{Max}_{i=1}^{Q}(B_i))^t$
- That is … Q-wise = AC
- t-wise is expensive and benefits are not clear

Input Space Partitioning Summary

- Fairly easy to apply, even with no automation
- Convenient ways to add more or less testing
- Applicable to all levels of testing – unit, class, integration, system, etc.
- Based only on the input space of the program, not the implementation

Esempio: NextDate

- NextDate è una funzione con tre variabili: month, day, year. Restituisce la data del giorno successivo alla data di input tra gli anni 1840 e 2040
- Specifica sommaria:
  - se non è l'ultimo giorno del mese, la funzione incrementa semplicemente il giorno.
  - Alla fine del mese il prossimo giorno è 1 e il mese è incrementato.
  - Alla fine dell'anno, giorno e mese diventano 1 e l'anno è incrementato.
  - Considerare il fatto che il numero di giorni del mese varia con il mese e l'anno bisestile

NextDate: Classi di Equivalenza

- M1 = {mese: il mese ha 30 giorni}
- M2 = {mese: il mese ha 31 giorni}
- M3 = {mese: il mese è Febbraio}
- D1 = {giorno: 1<= giorno <= 28}
- D2 = {giorno : giorno = 29}
- D3 = {giorno : giorno = 30}
- D4 = {giorno : giorno = 31}
- Y1 = {anno: anno = 1900}
- Y2 = {anno: ((anno != 1900) AND (anno mod 4 = 0)}
- Y3 = {anno: anno mod 4 != 0}

NextDate Test Cases (1)

- WECT: 4 test cases

Case ID:	Month	Day	Year	Output
WE1:	6	14	1900	6/15/1900
WE2:	7	29	1912	7/30/1912
WE3:	2	30	1913	Error
WE4:	6	31	1900	Error

- SECT: 36 test cases >> WECT

NextDate: SECT 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

- La copertura **N-WECT** è la minima copertura considerabile. Non garantisce un livello adeguato di test per la maggior parte delle situazioni.
  - Non considera l'eventuale correlazione tra i parametri di input (*single fault assumption*).
- La copertura SECT garantisce un livello di test troppo approfondito.
  - E' più adeguata nel caso in cui vi sia correlazione tra i parametri di input (ne analizza tutte le combinazioni).

Testing dei valori limite (boundary values)

- Con le classi di equivalenza si partiziona il dominio di ingresso assumendo che il comportamento del programma su input della stessa classe è simile
- Con i criteri WECT e SECT, si testano tipicamente i valori medi di ogni classe di equivalenza.
- Tipici errori di programmazione capitano però spesso al limite tra classi diverse
- Il **testing dei valori limite** si focalizza su questo aspetto
- Serve a estendere la tecnica precedente

Analisi dei valori limite

- Consideriamo una funzione *foo(x1, x2)*
- Classi di equivalenza:
  - $CE_{X1\text{-}1}$: a <= x1 <= b (valida)
  - $CE_{X1\text{-}2}$: x1 < a (non valida)
  - $CE_{X1\text{-}3}$: x1 > b (non valida)
  - $CE_{X2\text{-}1}$:c <= x2 <= d (valida)
  - $CE_{X2\text{-}2}$: x2 < c (non valida)
  - $CE_{X2\text{-}3}$: x2 > d (non valida)

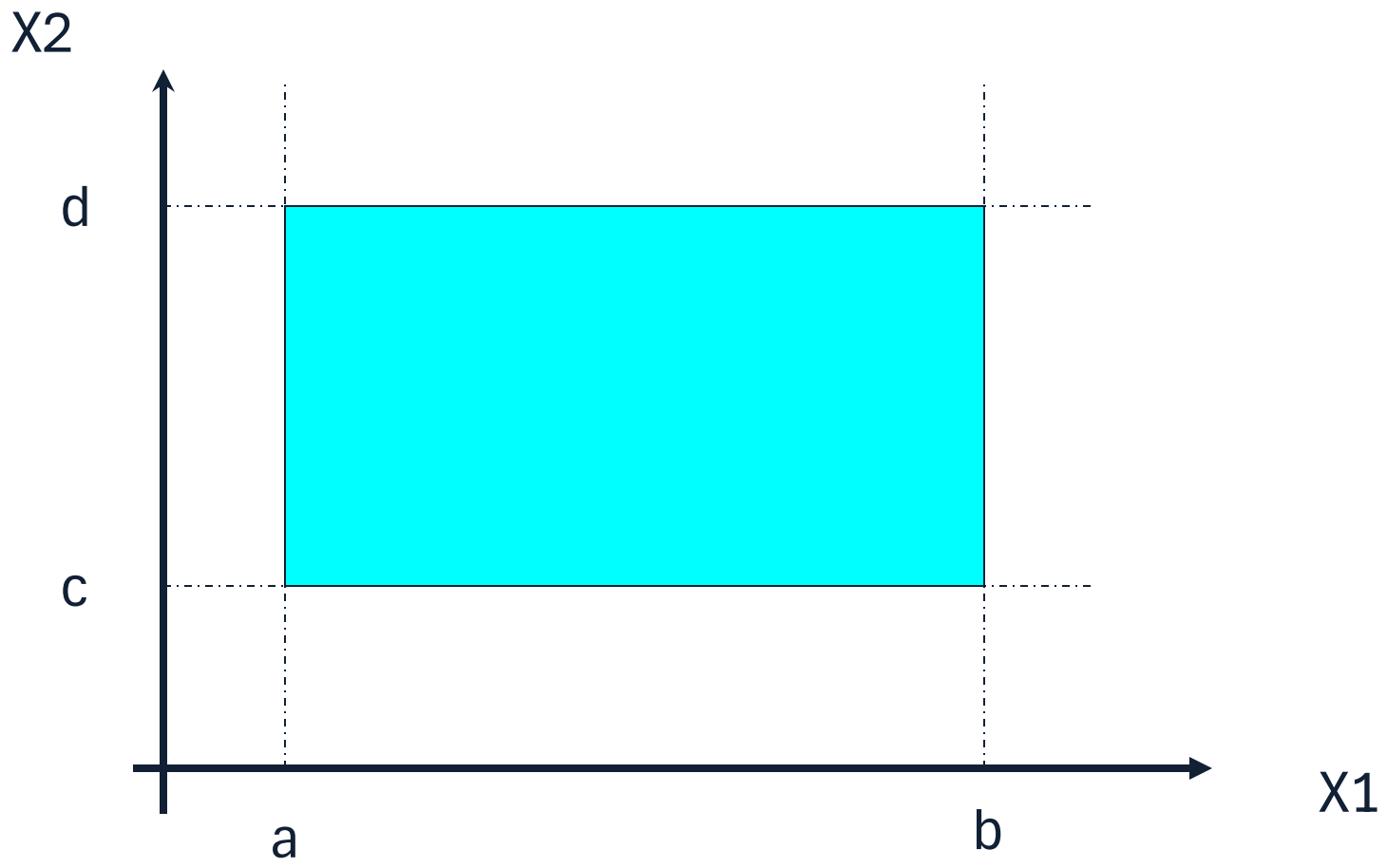
- Il metodo si focalizza sui limiti dello spazio di input per individuare i casi di test
- Il razionale (supportato da studi empirici) è che gli errori tendono a capitare vicino ai valori limite delle variabili di input

Idee di base

- Verificare I valori della variabile di input al minimo, immediatamente sopra il minimo, un valore intermedio (nominale), immediatamente sotto il massimo e al massimo, per ogni classe di equivalenza
  - Convenzione: *min, min+, nom, max-, max*
- Approccio: $\forall$ variabile di input v_i

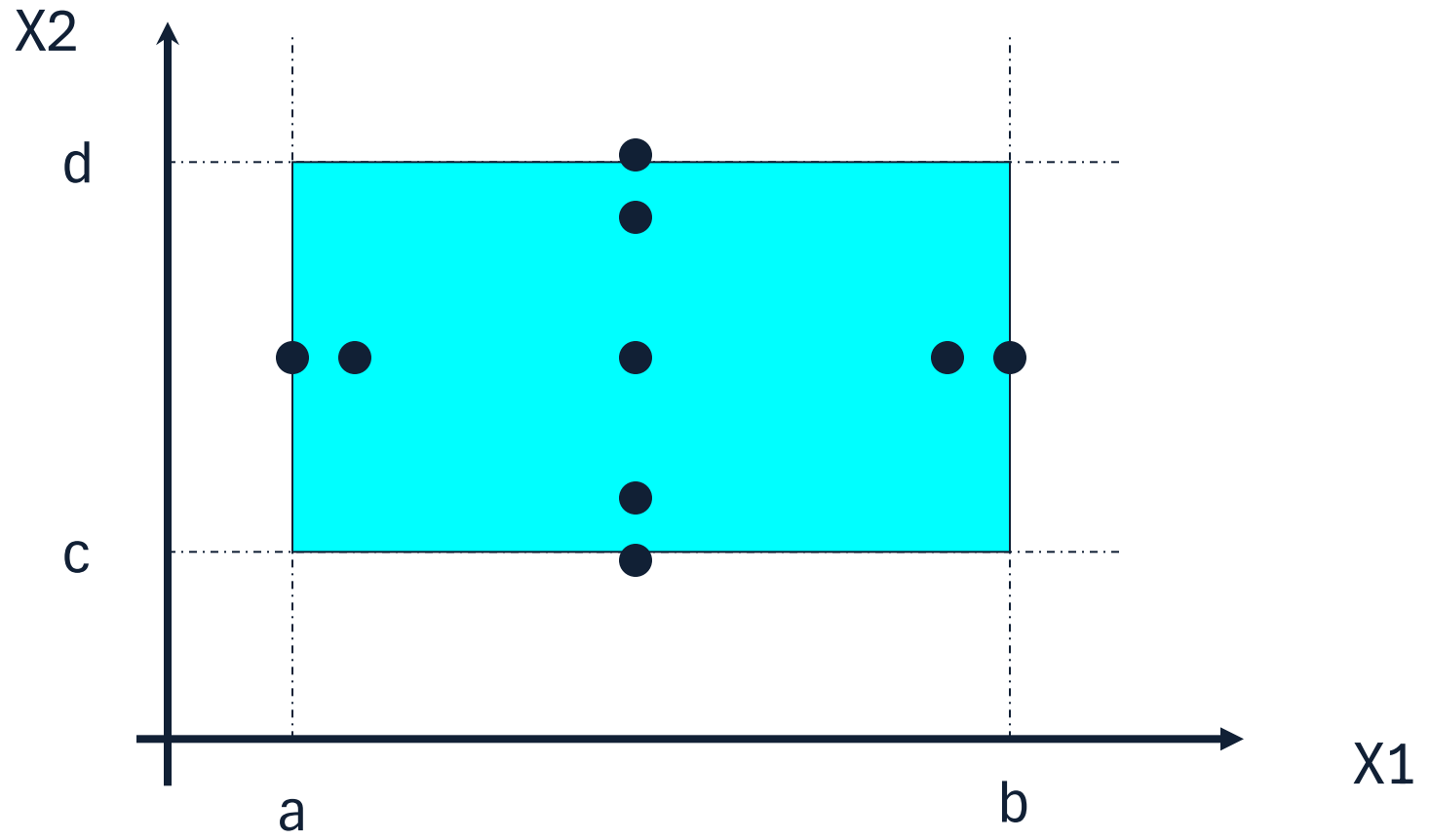
  v_i spazia tra questi 5 valori, mentre tutte le altre sono mantenute al loro valore nominale.

ISP dell'input della Funzione F



Boundary Analysis Test Cases

- Test set1 ={<x1nom, x2min>, <x1nom, x2min+>, <x1nom, x2nom>, <x1nom, x2max->, <x1nom, x2max>, <x1min, x2nom,>, <x1min+, x2nom,>, <x1max-, x2nom>, <x1max, x2nom>}

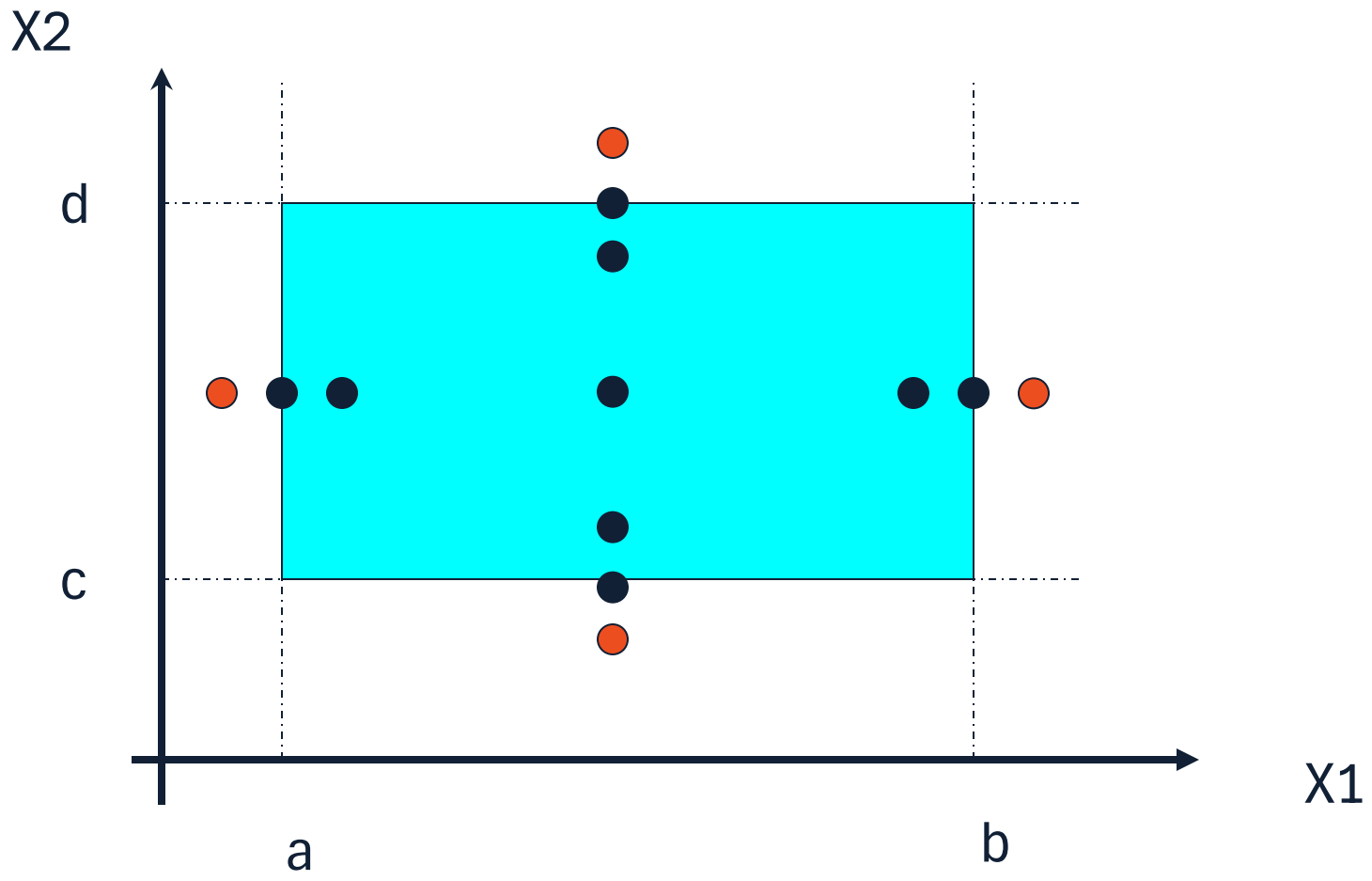


Caso Generale e Limitazioni

- Una funzione con n variabili richiede 4n + 1 casi di test
- Funziona bene con variabili che rappresentano quantità fisiche limitate
- Non considera la natura della funzione e il significato delle variabili
- Tecnica rudimentale che tende al test di robustezza

Test di Robustezza

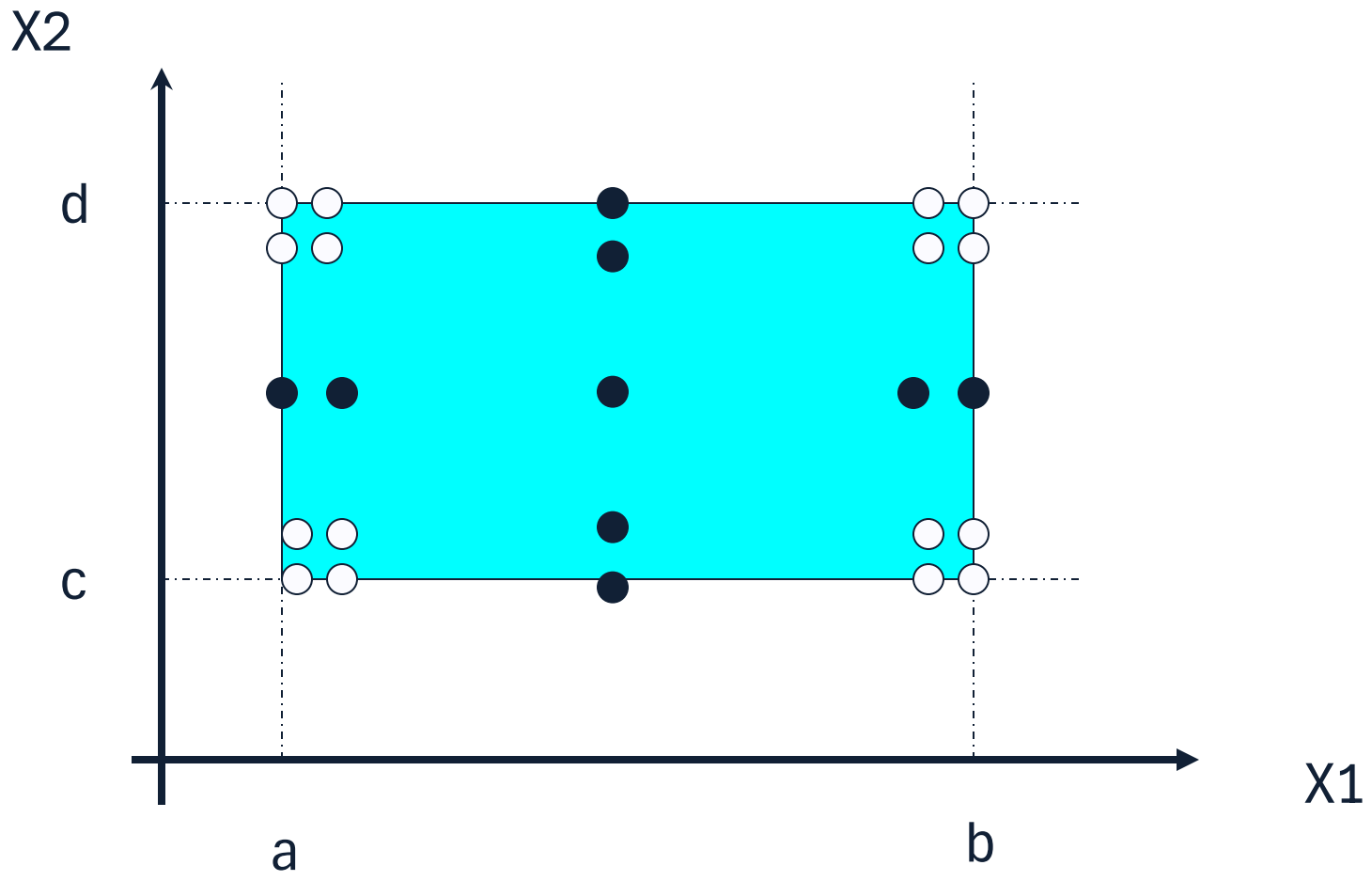
Esegue 6n+1 casi di test, date n variabili di input



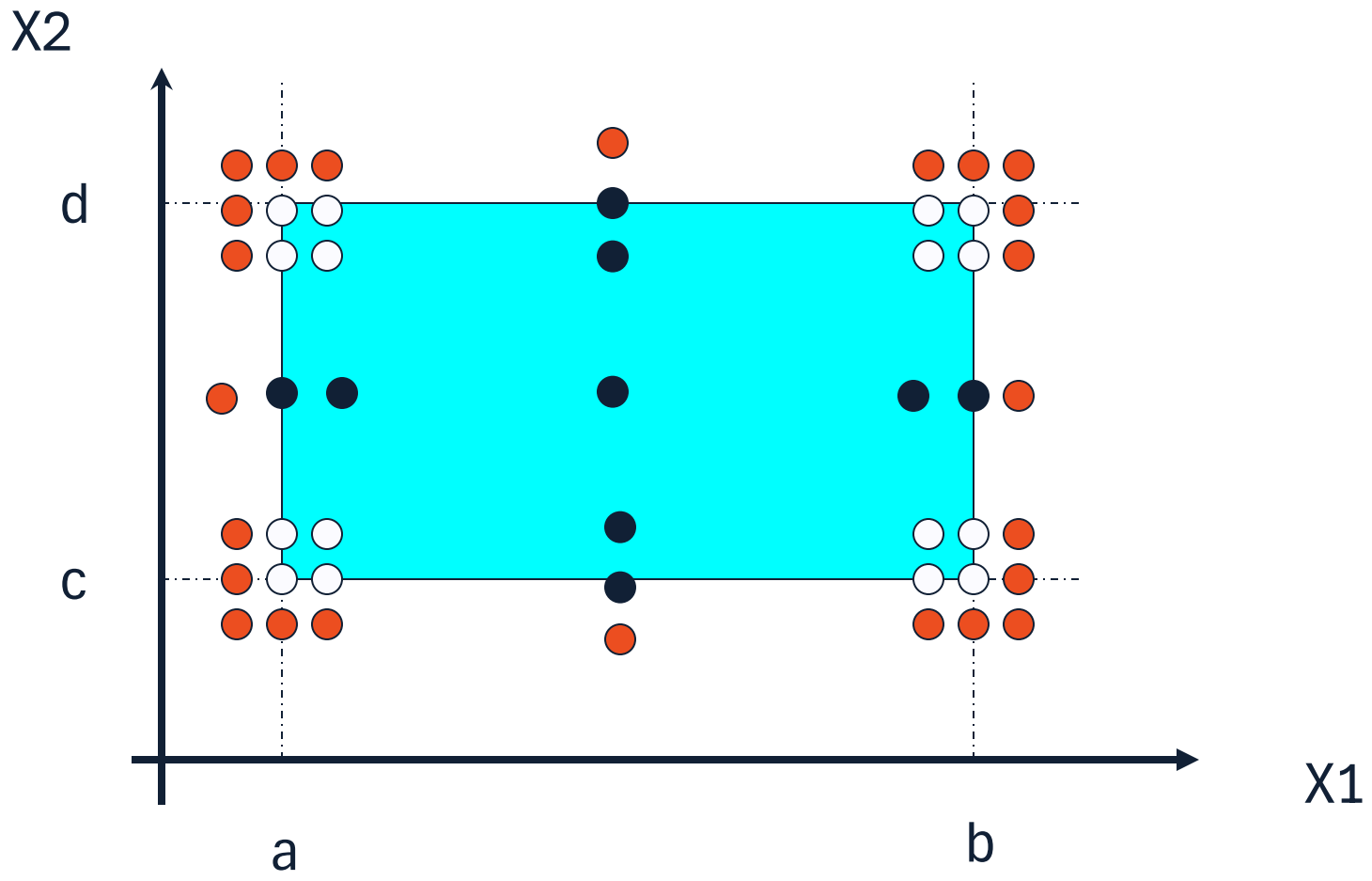
Worst Case Testing (WCT)

- La tecnica dei valori limite estende notevolmente la copertura rispetto al SECT, ma ha un problema:
  - Non testa mai più variabili **contemporaneamente** ai valori limite!
- WCT estende Boundary, testando il prodotto cartesiano di {min, min+, nom, max-, max}, per tutte le variabili di input, per tutte le classi di equivalenza.
- Più profonda del boundary testing, ma molto più costosa: 5^n test cases
- Buona strategia quando le variabili fisiche hanno numerose interazioni e quando le failure sono costose
- E in più: Robust Worst Case Testing

WCT per 2 variabili non indipendenti



Robust WCT per 2 variabili



Gerarchia

- Per n parametri di input:
  - Boundary Value testing of n inputs : $4n + 1$
  - Robustness testing of n inputs : $6n + 1$
  - Worst case for boundary value : 5^n
  - Worst case for robustness : 7^n

- Boundary Value is a subset of Robustness
- Worst case for boundary value is a subset of worst case of robustness

```java
// Returns a list of divisors of number
static List<Integer> getDivisors(int number){
```

Testing White Box

Testing Strutturale

- La definizione dei casi di test e dell'oracolo è basata su caratteristiche particolari della struttura interna dell'unità
- Si selezionano quei test che esercitano **tutte** le *strutture* interne del codice
  - **In base a cosa si intenda per *struttura* si avranno diverse strategie**
- Non è scalabile (usato soprattutto a livello di unità o sottosistema)
- Attività complementare al testing Black-Box
- Non può rilevare difetti che dipendono dalla mancata implementazione di alcune parti della specifica

Un modello di rappresentazione dei programmi (1/2)

Il *Grafo del Flusso di Controllo* di un programma P è una quadrupla GFC(P) = (N, E, n_i, n_f) dove:

(N,E) è un grafo diretto con archi etichettati

$$ni \in N, nf \in N, N - \{ni, nf\} = Ns \cup Np$$

N_s e N_p sono insiemi disgiunti di nodi, ossia $Ns \cap Np = \varnothing$
che rappresentano rispettivamente istruzioni e predicati

$$E \subseteq (N - nf) \times (N - ni) \times \{true, false, uncond\}$$

rappresenta la relazione di flusso di controllo

Un modello di rappresentazione dei programmi (2/2)

- ni ed nf sono detti nodo iniziale e nodo finale

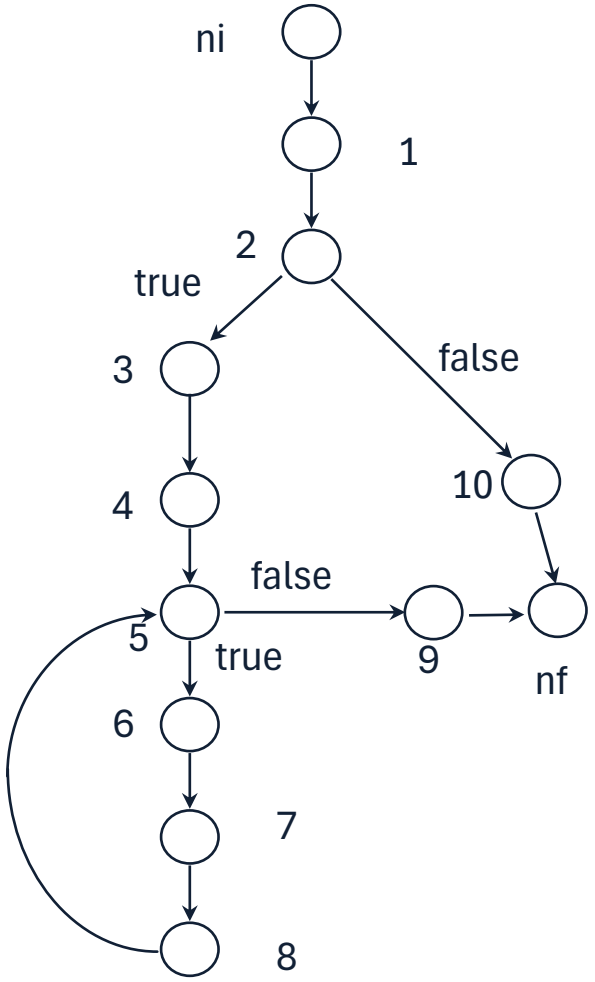
- Un nodo in $Ns \cup \{ni\}$ ha un solo successore immediato ed il suo arco è etichettato con uncond.

- Un nodo in Np ha due successori immediati e i suoi archi uscenti sono etichettati rispettivamente con true e false

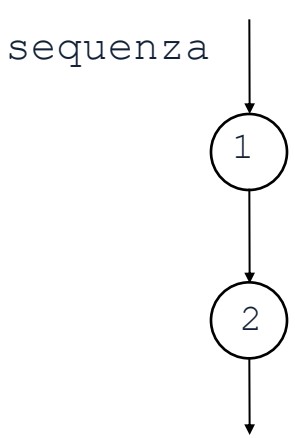
- Un GFC(P) e' ben formato se esiste un cammino dal nodo iniziale ni ad ogni nodo in N-{ni} e da ogni nodo in N-{nf} al nodo finale nf
- Diremo semplicemente cammino o cammino totale un cammino da ni a nf

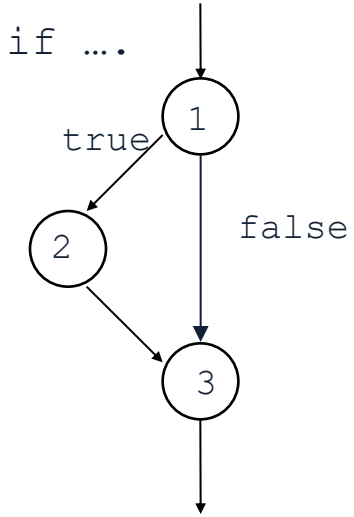
Esempio di GFC

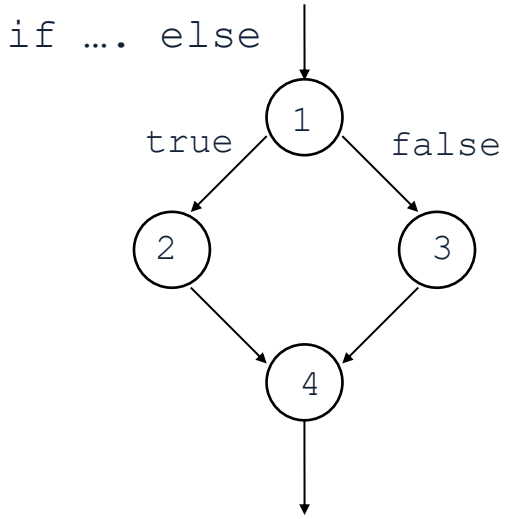
```
int foo(int x) {
1 int y,n;
2 if (x>0) {
3      n = 1;
4      y = 1;
5      while (x>1) {
6            n = n + 2;
7            y = y + n;
8            x = x - 1;
       }
9      return y;
   }
10 return 0;
}
```

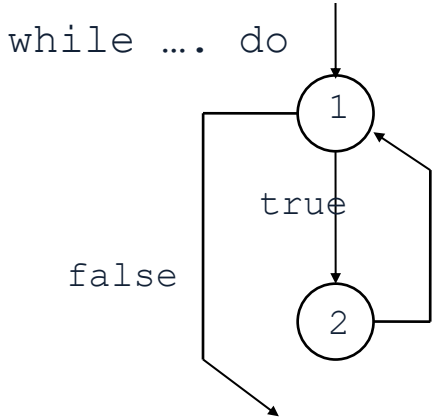


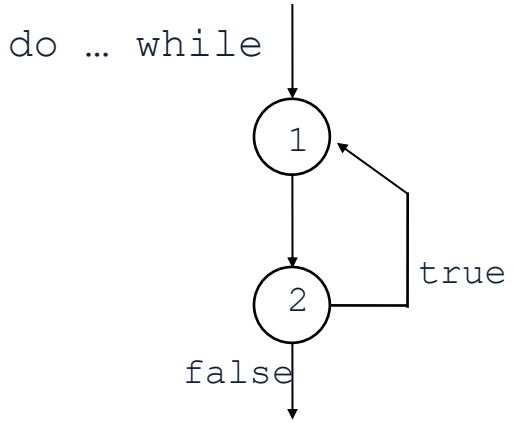
Costruzione del control flow graph per programmi strutturati









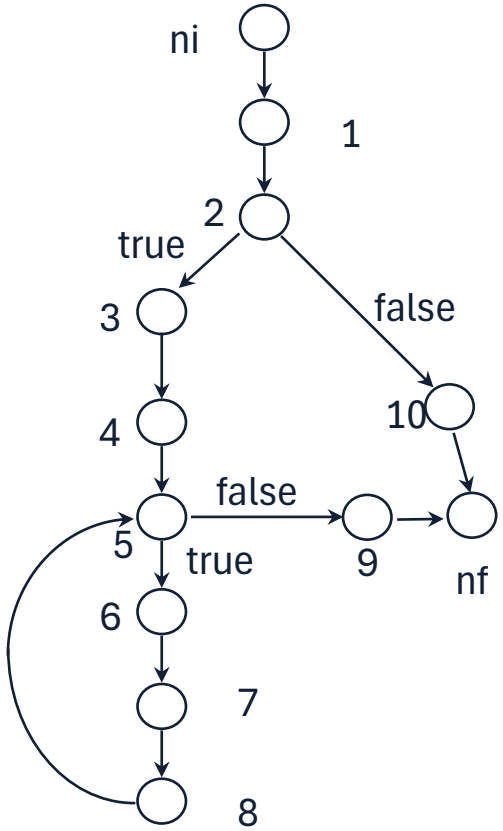


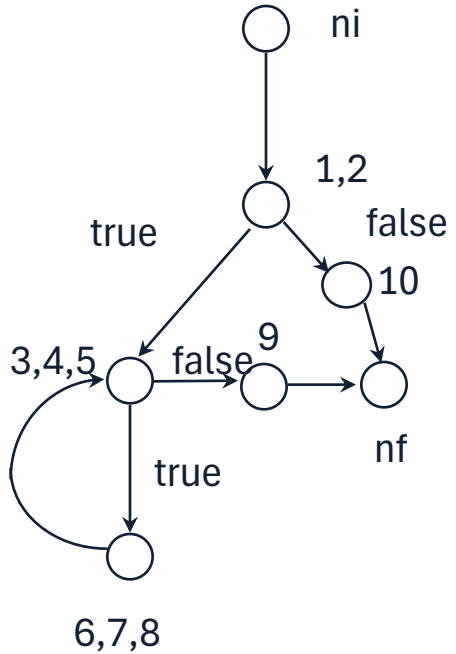
Semplificazione di un CFG

- Sequenza di nodi possono essere collassate in un solo nodo, purché nel grafo semplificato vengano mantenuti tutti i branch (punti di decisione e biforcazione del flusso di controllo)
- Tale nodo collassato può essere etichettato con i numeri dei nodi in esso ridotti

Esempio (con semplificazione)

```
int foo(int x) {
1 int y,n;
2 if (x>0) {
3       n = 1;
4       y = 1;
5       while (x>1) {
6               n = n + 2;
7               y = y + n;
8               x = x - 1;
        }
9       return y;
   }
10 return 0;
}
```





Tecniche di Testing Strutturale

- In generale fondate sull'adozione di criteri di copertura degli elementi che compongono la struttura dei programmi
  - COPERTURA: definizione di un insieme di casi di test in modo tale che gli oggetti di una definita classe (es. strutture di controllo, istruzioni, archi del GFC, predicati,..etc.) siano attivati almeno una volta nell'esecuzione dei casi di test
  - Definizione di una metrica di copertura:

  - Test Effectiveness Ratio (TER) = # elementi coperti / # elementi totale

Selezione di casi di test

- Problema: come selezionare i casi di test per il criterio di copertura adottato ?
  - Quale struttura vado a testare? Branch? Nodi?
  - Ogni caso di test corrisponde all'esecuzione di un particolare cammino sul GFC di un programma P
  - Individuare i cammini che ci garantiscono il livello di copertura desiderato

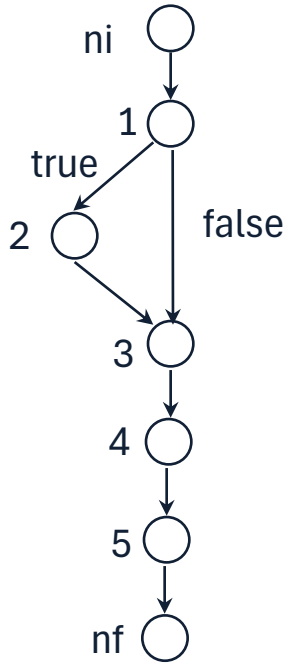
Copertura dei nodi (statement)

- Dato un programma P, viene definito un insieme di test case la cui esecuzione implica l'attraversamento di tutti i nodi di GFC(P), ovvero l'esecuzione di tutte le istruzioni di P
  - Test Effectiveness Ratio (TER)

$$\text{TER} = \frac{\text{n.ro di statement eseguiti}}{\text{n.ro di statement totale}}$$

Node Coverage: Esempio

```
 void statement(double x, double y)
    {
1.    if (x != 0)
2.              x = x + 10;
3.    y = y/x;
4.    cout << x;
5.    cout << y;
     }
```



Node Coverage: Esempio

- copertura del 100%: cammino 1, 2, 3, 4, 5
- Per coprire tutti i nodi → qualunque x != 0
- input data per caso di test:
  - y: qualsiasi valore;
  - x: valore diverso da 0
- NB: failure per x = 0 e x = -10

- E' un tipo di test che non garantisce di aver percorso tutte le "strade" (rami) almeno una volta
- Un test che come prima esegue tutti i comandi, ma non tutti i rami
- Altre soluzioni → Branch Coverage

Copertura delle decisioni (branch)

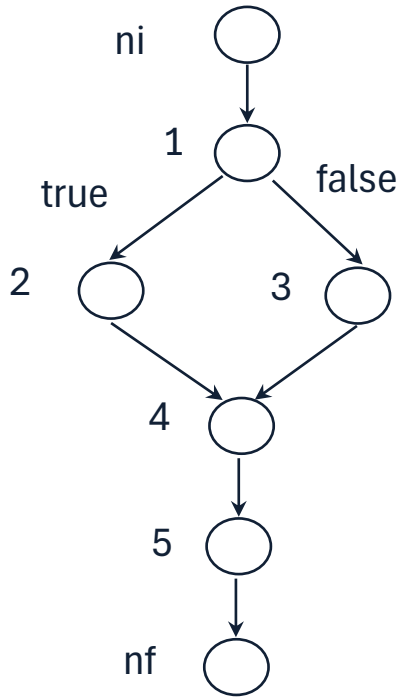
- Dato un programma P, viene definito un insieme di test case la cui esecuzione implica l'attraversamento di tutti i rami di GFC(P), ovvero l'esecuzione di tutte le decisioni di P
  - Test Effectiveness Ratio (TER)

$$\text{TER} = \frac{\text{n.ro di branch eseguiti}}{\text{n.ro di branch totali}}$$

- NB: la copertura delle decisioni implica la copertura dei nodi

Branch Coverage: Esempio

```
 void branch(double x, double y)
    {
1.     if (x == 0 || y > 0)
2.              y = y / x;
       else
3.              x = y + 2/x;
4.     cout << x;
5.     cout << y;
      }
```



Branch Coverage: Esempio

- *cammini*:
  - a) 1, 2, 4, 5
  - b) 1, 3, 4, 5
- *path conditions*:
  - a) *X* == 0 || *Y* > 0
  - b) *X* != 0 && *Y* <= 0
- Possibili input data:
  - a) y > 0, x != 0
  - b) y<= 0, x != 0

- *NB: failure per x = 0*
- Altre soluzioni → Condition Coverage

Copertura di decisioni e condizioni

- Dato un programma P, viene definito un insieme di test case la cui esecuzione implica l'esecuzione di tutte le decisioni e di tutte le condizioni caratterizzanti le decisioni in P:
  - Condition coverage: Per ogni decisione vengono considerate tutte le combinazioni di condizioni

Copertura di decisioni e condizioni

- **Modified condition coverage:** For safety-critical applications (e.g. for avionics software) it is often required that modified condition/decision coverage (MC/DC) is satisfied.
- This criteria extends condition/decision criteria with requirements that each condition should affect the decision outcome independently.
- Per ogni condizione vengono considerate solo le combinazioni di valori per le quali una delle condizioni determina il valore di verità della decisione
  - Riduzione dei casi di test ...
  - It is used in the standard DO-178B
    - The FAA accepts use of DO-178B as a means of certifying software in avionics

Esempio: Modified Condition Coverage

- Esempio : $A \wedge (B \vee C)$

	ABC	Res.	Corr. False Case
1	TTT	T	A (5)
2	TTF	T	A (6), B (4)
3	TFT	T	A (7), C (4)
4	TFF	F	B (2), C (3)
5	FTT	F	A (1)
6	FTF	F	A (2)
7	FFT	F	A (3)
8	FFF	F	-

Prendere una coppia per ogni condizione:

- A : (1,5), opp. (2,6), opp. (3,7)
- B : (2,4)
- C : (3,4)

Due insiemi minimi per coprire il modified condition criterion:

- (2,3,4,6) or (2,3,4,7)

4 casi di test test invece di 8 per tutte le possibili combinazioni

Copertura dei cammini

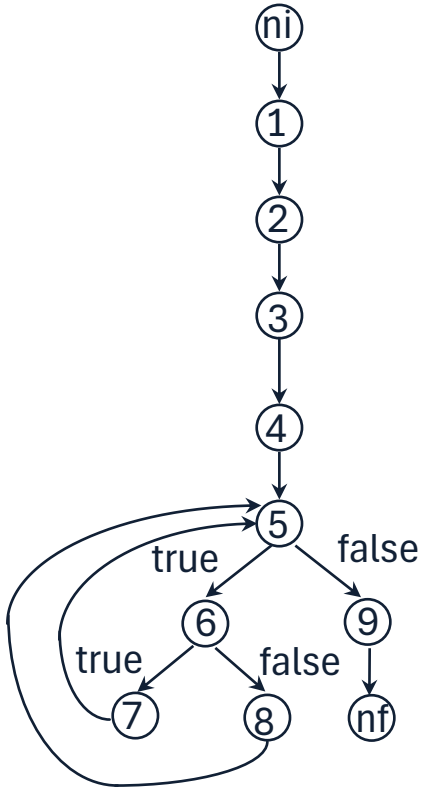
- Dato un programma P, viene definito un insieme di test case la cui esecuzione implica l'attraversamento di tutti i cammini di GFC(P)
  - Test Effectiveness Ratio (TER)

$$\text{TER} = \frac{\text{n.ro di cammini eseguiti}}{\text{n.ro di cammini totali}}$$

- Problemi:
  - numero di cammini infinito (o comunque elevato…)
  - infeasible path

Esempio

```
void gcd() {
     int x, y, a, b;
1.   cin >> x;
2.   cin >> y;
3.   a = x;
4.   b = y;
5.   while (a != b)
6.           if(a > b)
7.             a = a - b;
8.           else  b = b - a;
9.   cout << a;
}
```



I cicli possono portare a cammini infiniti:
1, 2, 3, 4, 5, 6, 7, 5, 6, 7, ...

Copertura dei cammini: soluzioni

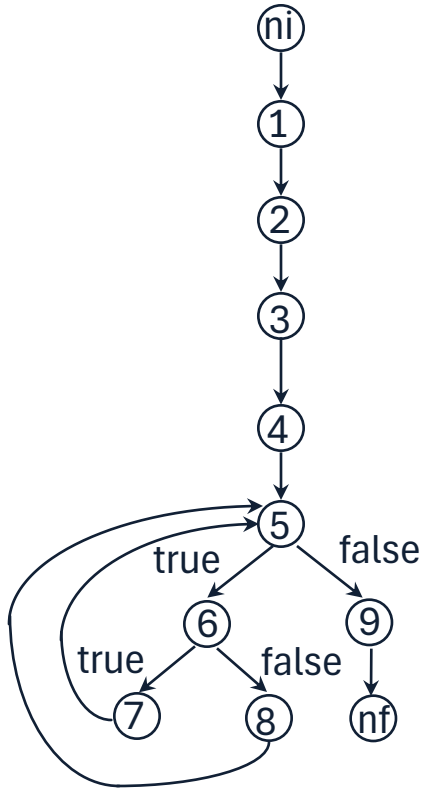
- Un numero di cammini infinito implica la presenza di circuiti
  - NB: il numero dei cammini elementari (privi di circuiti) in un grafo è finito
- Soluzione: limitare l'insieme dei cammini
  - Criterio di n-copertura dei cicli
  - Metodi dei cammini linearmente indipendenti (Mc Cabe)
  - …

Criterio di n-copertura dei cicli

- Si seleziona un insieme di test case che garantisce l'esecuzione dei cammini contenenti un numero di iterazioni di ogni ciclo non superiore ad n (ogni ciclo deve essere eseguito da 0 ad n volte)
- Al crescere di n può diventare molto costoso
  - Caso pratico n = 2 (ogni ciclo viene eseguito 0 volte, 1 volta , 2 volte)
- NB: il criterio di 1-copertura dei cicli (n = 1) implica il criterio di copertura dei branch

Esempio: 2-copertura dei cicli

```
int gcd() {
   int x, y, a, b;
1. cin >> x;
2. cin >> y;
3. a = x;
4. b = y;
5. while (a != b)
6.       if(a > b)
7.                a = a - b;
8.       else  b = b - a;
9. return y;
}
```



0 volte:

1, 2, 3, 4, 5, 9

1 volta:

1, 2, 3, 4, 5, 6, 7, 5, 9

1, 2, 3, 4, 5, 6, 8, 5, 9

2 volte:

1, 2, 3, 4, 5, 6, 7, 5, 6, 7, 5, 9

1, 2, 3, 4, 5, 6, 7, 5, 6, 8, 5, 9

1, 2, 3, 4, 5, 6, 8, 5, 6, 8, 5, 9

1, 2, 3, 4, 5, 6, 8, 5, 6, 7, 5, 9

fault non individuato con il cammino

Confronto tra White & Black-box Testing

- White-box Testing:
  - Un numero potenzialmente infinito di path da testare
  - Testa cosa è stato fatto, non cosa doveva essere fatto
  - Non può scoprire Use Case mancanti
- Black-box Testing:
  - Potenzialmente c'è un'esplosione combinatoria di test cases
  - Non permette di scoprire use cases estranei ("features")

- Entrambi sono necessari
- Sono agli estremi di un ipotetico testing continuum.
- Qualunque test case viene a cadere tra loro, in base a:
  - Numero di path logici possibili
  - Natura dei dati di input
  - Complessità di algoritmi e strutture dati

Verification & Validation

Integration testing

- Quando ogni componente è stata testata in isolamento, possiamo integrarle in sottosistemi più grandi.
- L'Integration testing mira a rilevare errori 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

Strategie di Integration testing

- 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

- Ognuna di queste strategie è stata originariamente concepita per una decompisizione gerarchica del sistema
  - ogni componente appartiene ad una gerarchia di layer, ordinati in base all’associazione “Call”

Functional Testing

Functional Testing e Requisiti

- Veriicare che tutti I requisiti funzionali siano stati implementati correttamente
- Impatto dei Requisiti sul testing del sistema:
  - Più espliciti sono i requisiti, più facile sono da testare.
  - La qualità degli use case influenza il Functional Testing

Functional Testing

- Goal: trovare differenze tra i requisiti funzionali e le funzionalità realmente offerte dal sistema
- I test case sono progettati dal documento dei requisiti e si focalizza sulle richieste e le funzioni chiave
- Il sistema è trattato come un black box.
- I test case per le unit possono essere riusati, ma devono essere scelti quelli rilevanti per l'utente finale e che hanno una buona probabilità di riscontrare un fallimento.

System Testing

System Testing

- Unit testing e Integration testing si focalizzano sulla ricerca di failures 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:
  - Performance Testing
  - Pilot Testing
  - Acceptance Testing
  - Installation Testing

Performance Testing

- Goal: spingere il sistema oltre i suoi limiti!
- Testare come il sistema si comporta quando è sovraccarico.
  - Possono essere identificati colli di bottiglia? (I primi candidati ad ad essere riprogettati)
- Tenta ordini di esecuzioni non usuali
  - Es: Invocare una receive() prima di una 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.
- Strumenti: jMeter

Pilot testing

- Primo test pilota, sul campo, del sistema
- Nessuna linea guida o scenario 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
  - Nuovo paradigma ampiamente utilizzato con la distribuzione del software tramite Internet
  - Offre il software a chiunque che è interessato a testarlo

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 competitore
  - **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 function 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

Usability testing

- Testa la comprensibilità del sistema da parte dell'utente
- I rappresentanti dei potenziali utenti trovano problemi "usando" le interfacce utente o una loro simulazione
  - "look and feel",
  - layout geometrico delle schermate,
  - sequenza delle interazioni
- Le tecniche sono basate sull'approccio degli esperimenti controllati:
  - Gli sviluppatori prima selezionano un insieme di obiettivi
  - In una serie di esperimenti, viene chiesto ai partecipanti di eseguire una serie di task sul sistema
  - Gli sviluppatori osservano i partecipanti e raccolgono una serie di informazioni oggettive
    - Time-to-complete di un task
    - Numero di errori
  - e soggettive
    - Questionari di gradimento
    - Think aloud

Gestione del Testing

I 4 passi del testing

- 1. Scegliere cosa deve essere testato
  - Completezza dei requisiti
  - Testare l'affidabilità del codice
  - ...
- 2. Decidere come deve essere testato
  - Code inspection
  - Black-box, white box, ...
- 3. Definire i test cases
  - Un insieme di test e/o situazioni utili a stressare il componente/sistema
- 4. Creare il test oracle
  - Necessario per poter svolgere i test

Piano di testing

- Un “piano di testing” è un documento che prevede i seguenti elementi:
  1. Introduzione
     - Descrizione degli obiettivi
  2. Relazioni con altri documenti
     - Come e dove sono specificati requisiti funzionali e non funzionali, eventuale glossario
  3. System Overview
     - Dettagli sul sistema e sul livello di granularità che si vuole testare
  4. Caratteristiche da testare/escludere
     - Considerando solo aspetti funzionali, contiene un elenco di tutte le caratteristiche che saranno testate e quelle che saranno escluse
  5. Criteri di Pass/Fail
     - Eventuali criteri di successo/fallimento (es. Errore inferiore dello 0,0001%)

Piano di testing

6. Approcci
   - Descrive le strategie di testing che saranno utilizzate
7. Risorse
   - Elenco dei requisiti hardware/software/ambientali per poter effettuare il test
8. Test cases
   - Il cuore del documento. Contiene l'elenco dei casi di test
9. Testing schedule
   - Scheduling temporale e di risorse per il testing

Definizioni di base sui Test Case

- Un **Test Case** è un insieme di input e di risultati attesi che stressano una componente con lo scopo di causare fallimenti e rilevare fault.
- Ha 5 attributi:
  - **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

1.1.1 Test delle funzionalità xxxxxxx

TEST ID	1	
TEST NAME	*NOME TEST*	
TEST DESCRIPTION	*DESCRIZIONE……*	
INPUT	**RISULTATO DESIDERATO**	**RISULTATO OTTENUTO**
LOCATION	*CARATTERISTICHE DELLA CONFIGURAZIONE DI TEST*	
NOTE	*EVENTUALI NOTE*	

2.4.1 Test delle funzionalità di Login

TEST ID	1	
TEST NAME	*Test Login*	
TEST DESCRIPTION	*Obiettivo di questo test è verificare la funzionalità di login*	
INPUT	**RISULTATO DESIDERATO**	**RISULTATO OTTENUTO**
Inserire “User1” nel campo “Nome Utente”, “1234” nel campo Password e premere OK	Login effettuato, visualizzata la home page dell’utente loggato	
Inserire “User1” nel campo “Nome Utente”, “12345” nel campo Password e premere OK	Login fallito, visualizzato messaggio “Nome utente o Password errati”	
Inserire “User12” nel campo “Nome Utente”, “1234” nel campo Password e premere OK	Login fallito, visualizzato messaggio “Nome utente o Password errati”	
Inserire “User12” nel campo “Nome Utente”, “12354” nel campo Password e premere OK	Login fallito, visualizzato messaggio “Nome utente o Password errati”	
Premere OK senza inserire nulla	Login fallito, visualizzato messaggio “Inserire Nome utente e Password”	
…	…	
LOCATION	*CARATTERISTICHE DELLA CONFIGURAZIONE DI TEST*	
NOTE	*Il database deve contenere l’utente “User1”, con Password “1234”*	