UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

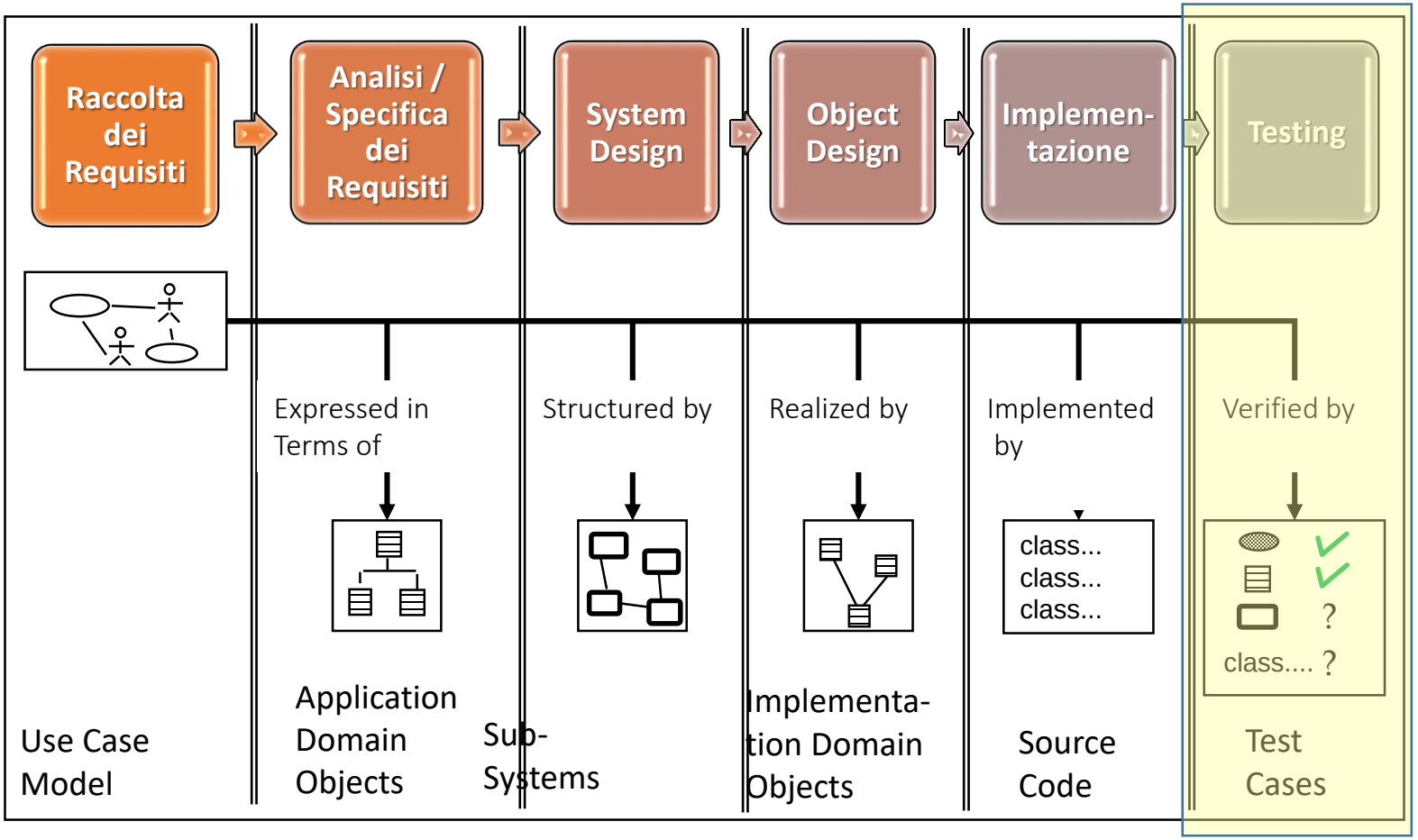
Ingegneria del Software – Definizione di Casi di Test

Prof. Sergio Di Martino

Obiettivi della lezione

- Comprendere le strategie per definire casi di test
  - Testing Black-Box
  - Testing White-Box

Ciclo di Vita del Software



Come testare una unità?

- Dopo la creazione dello scaffolding siamo pronti a testare l'unità.
- 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?

- Consider the following program:

```
Input A          // the program accepts any
Input B          // integer into A and B
Print A/B
```

- We can achieve complete coverage easily.
- Set A to 2 and B to 1 and you've covered every statement (and every branch) in this program.
- But this doesn't achieve complete testing.
  - What test is missing?
  - What bug was missed?

Measuring and achieving high coverage

- Coverage measurement is a good tool to show **how far** you are from complete testing.
- But it's a lousy tool for investigating how close you are to completion.
- Driving testing to achieve "high" coverage is likely to yield a mass of low-power tests.
  - People optimize what we measure them against, at the expense of what we don't measure.
    - For more on measurement distortion and dysfunction, read Bob Austin's book, Measurement and Management of Performance in Organizations.

Strategie di testing

- Come scegliamo i valori con cui testare i metodi di un'unità?
- Due strategie fondamentali:
  - **Black Box**: non considera la struttura del codice sorgente
  - **White Box**: si basa sulla struttura del codice sorgente

II Testing Black Box

Testing Funzionale

- La definizione del casi di test e dell'oracolo è fondata esclusivamente sulla base dei requisiti per l'unità da testare.
- L'unità per noi è una scatola nera (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

Criteri di copertura per testing funzionale

- Esempio:
  - Eseguire almeno una volta ogni funzionalità specificata nei documenti di progetto
  - Per ogni funzionalità, effettuare un numero di esecuzioni dedotte dai dati di ingresso e di uscita, da pre-condizioni e post-condizioni
- Occorre gestire il problema di trovare una opportuna copertura del dominio di input delle funzionalità
- Utilizzo delle **classi di equivalenza**

Le Classi di Equivalenza

Testing Black Box

Equivalence Partitioning

- Typically the universe of all possible test cases is so large that you cannot try them all
- You have to select a relatively small number of test cases to actually run
- Which test cases should you choose?
- **Equivalence partitioning** helps answer this question

Equivalence Partitioning

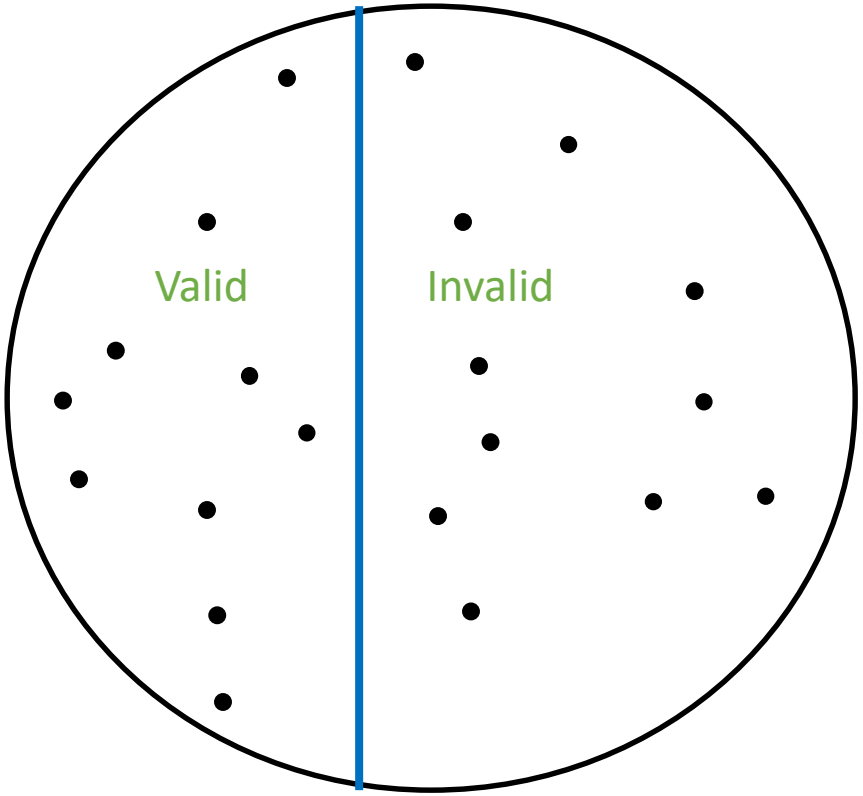
- Equivalence partitioning is a software testing technique that divides the input and/or output data of a software unit into partitions of data from which test cases can be derived.
- The equivalence partitions are usually derived from the requirements specification for input attributes that influence the processing of the test object.
- Test cases are designed to cover each partition at least once.

Suddivisione in classi di equivalenza

- **Idea**: se un caso di test non rileva fallimenti per un elemento della classe, 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 un suo rappresentante.
- La validità dell'assunto dipende dalla ragionevole progettazione della partizione del dominio in classi di equivalenza.

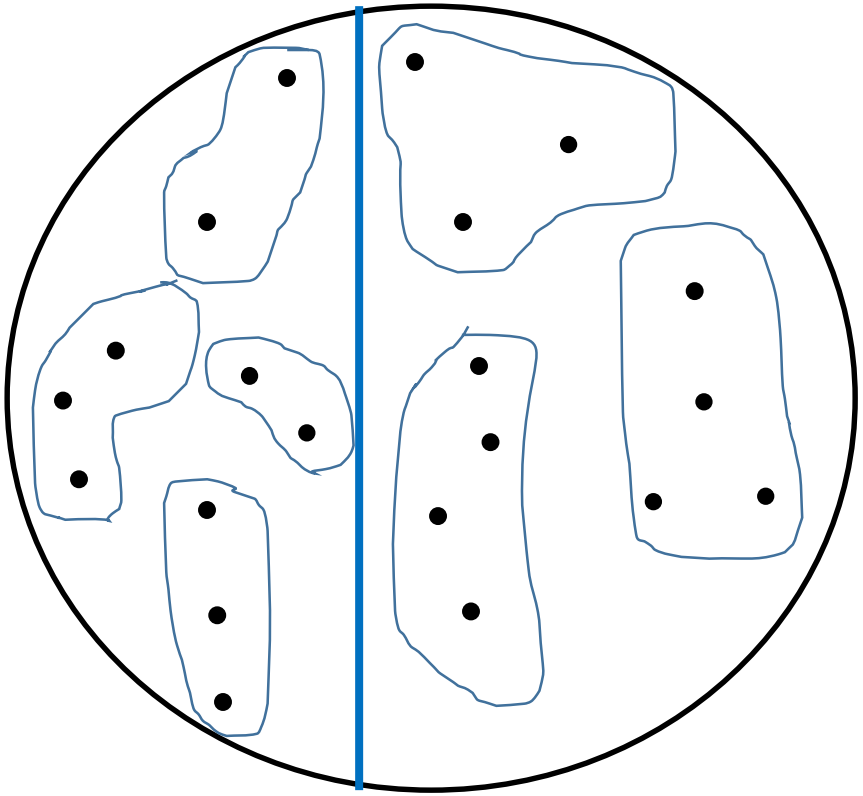
Equivalence Partitioning

- First-level partitioning: Valid vs. Invalid test cases



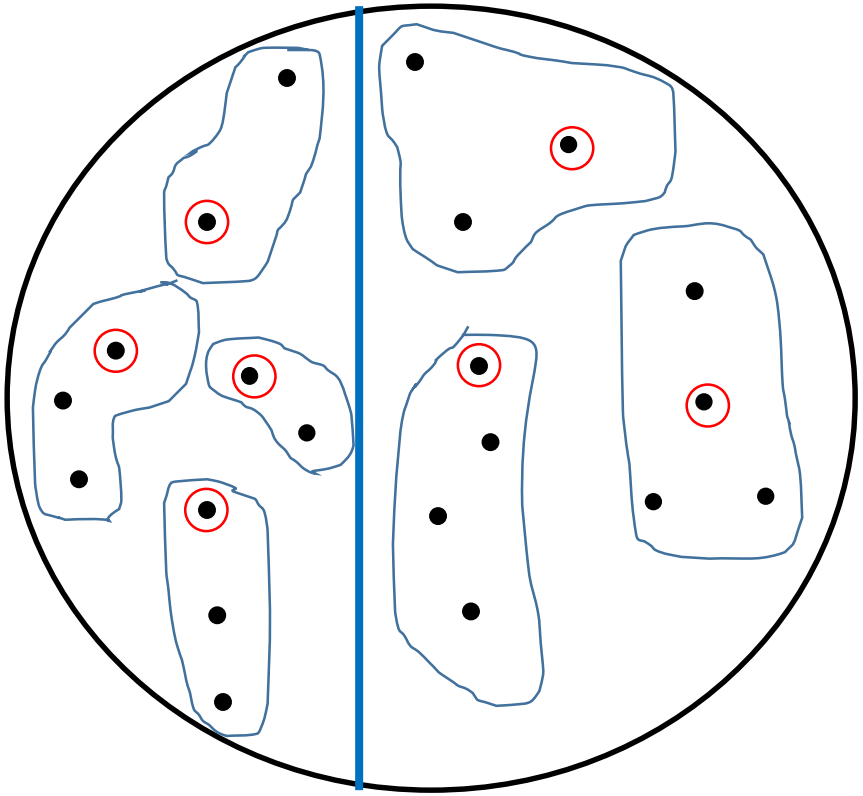
Equivalence Partitioning

- Partition valid and invalid test cases into equivalence classes



Equivalence Partitioning

- Create a test case for at least one value from each equivalence class



Come identificare le Classi di Equivalenza?

- Individuazione delle **caratteristiche** da partizionare
- Le caratteristiche sono **dimensioni nel dominio** della funzionalità oggetto di test
  - **Caratteristiche sintattiche derivanti dalla signature**
    - Una caratteristica per ogni parametro in ingresso della funzionalità
  - **Caratteristiche semantiche derivanti dalla specifica.**
    - Dipendono dalla funzionalità specifica
    - Spesso correlate ai parametri di uscita della funzionalità
    - Possono essere trasversali (correlandole) rispetto alle caratteristiche sintattiche.

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)

Esempio:
int triangolo(int a, int b, int c)

- **Specifica:** la funzione riceve in ingresso la dimensione dei tre lati e restituisce una classificazione del triangolo corrispondente (-1 non triangolo, 0 equilatero, 1 isoscele, 2 scaleno).
- Caratteristiche sintattiche
  - Parametro a, Dominio int, A1 = { a: a> 0} A2 = { a: a<= 0}
  - Parametro b, Dominio int, B1 = { b: b> 0} B2 = { b: b<= 0}
  - Parametro c, Dominio int, C1 = { c: c> 0} C2 = { c: c<= 0}
- Caratteristiche semantiche derivanti dalla specifica.
  - Tipo Triangolo, Dominio {NonTriangolo, Equilatero, IsosceleNonEquilatero, Scaleno}. D1 = { NonTriangolo}, D2 = {Equilatero}, D3 = {IsosceleNonEquilatero}, D4 ={Scaleno}

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

- Qualora la funzione da testare prenda un solo parametro in input, gli approcci precedenti sono sufficienti per definire tutti i casi di test.
- 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 diversi criteri di copertura con implicazioni diverse sul numero dei test generati.

Definizione dei casi di test dalle classi di equivalenza

- Supponiamo una funzione *void foo(a,b,c)*, che prenda tre variabili di input dei domini A, B, C
- Supponiamo di aver partizionato i domini in:
  - $A = A1 \cup A2 \cup A3$ dove $an \in An$
  - $B = B1 \cup B2 \cup B3 \cup B4$ dove $bn \in Bn$
  - $C = C1 \cup C2$ dove $cn \in Cn$

WECT Test Cases

- Weak Equivalence Class Testing - **WECT**: **Per ogni classe di equivalenza ci deve essere un Test case che usa un valore nominale 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)

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

Esempio 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	out
T1	3	3	3	0
T2	0	0	0	-1
T3	3	3	5	1
T4	3	4	5	2

SECT Test Cases

- Strong Equivalence Class Testing – **SECT**
- 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

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

Esempio SECT: int triangolo(int a, int b, int c)

Test	a	b	c	D	out
1	3	3	3	E	0
2	3	3	5	I	1
3	3	4	5	S	2
4	1	1	3	N	-1
5	-3	3	3	N	-1
6	-3	3	3	E	No
7	-3	3	3	I	No
8	-3	3	3	S	No
9	3	-3	3	N	-1
10	3	-3	3	E	No
11	3	-3	3	I	No
12	3	-3	3	S	No
13	3	3	-3	N	-1
14	3	3	-3	E	No
15	3	3	-3	I	No
16	3	3	-3	S	No

Test	a	b	c	D	out
17	-3	3	-3	N	-1
18	-3	3	-3	E	No
19	-3	3	-3	I	No
20	-3	3	-3	S	No
21	-3	-3	-3	N	-1
22	-3	-3	-3	E	No
23	-3	-3	-3	I	No
24	-3	-3	-3	S	No
25	-3	-3	3	N	-1
26	-3	-3	3	E	No
27	-3	-3	3	I	No
28	-3	-3	3	S	No
29	3	-3	-3	N	-1
30	3	-3	-3	E	No
31	3	-3	-3	I	No
32	3	-3	-3	S	No

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 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.
  - Richiede un numero minimo di casi di test pari al numero massimo di classi associato a una caratteristica.
- La copertura SECT garantisce un livello di test più approfondito.
  - E' più adeguata nel caso in cui vi sia correlazione tra i parametri di input (ne analizza tutte le combinazioni).
  - Richiede un numero minimo di casi di test pari al prodotto del numero di classi associato a ciascuna caratteristica (cardinalità del prodotto cartesiano).

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)

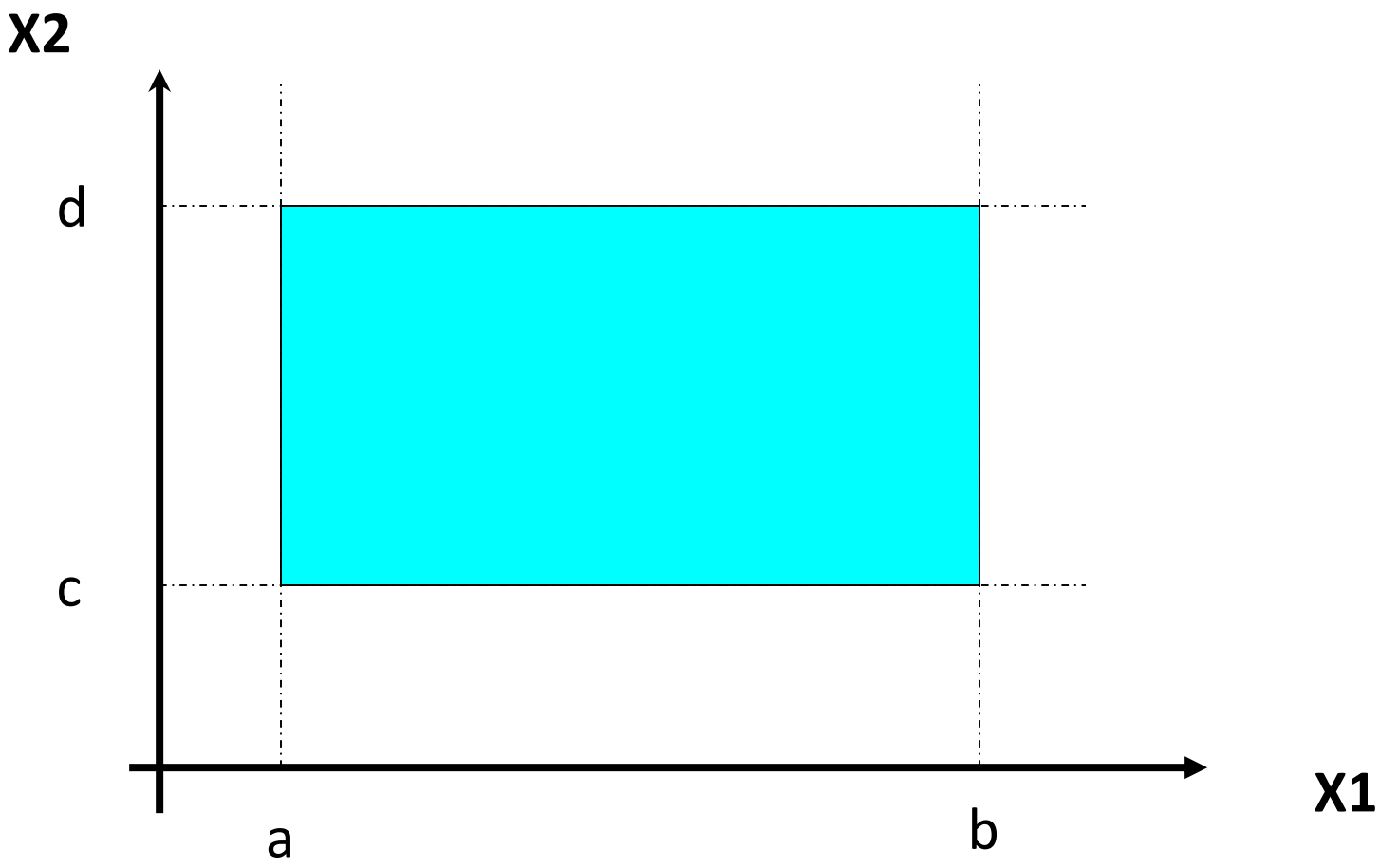
- Il metodo si focalizza sui limiti dello spazio di input per individuare i casi di test
- Il razionale (supportato da studi empirici) è che gli errori tendono a capitare vicino ai valori limite delle variabili di input

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

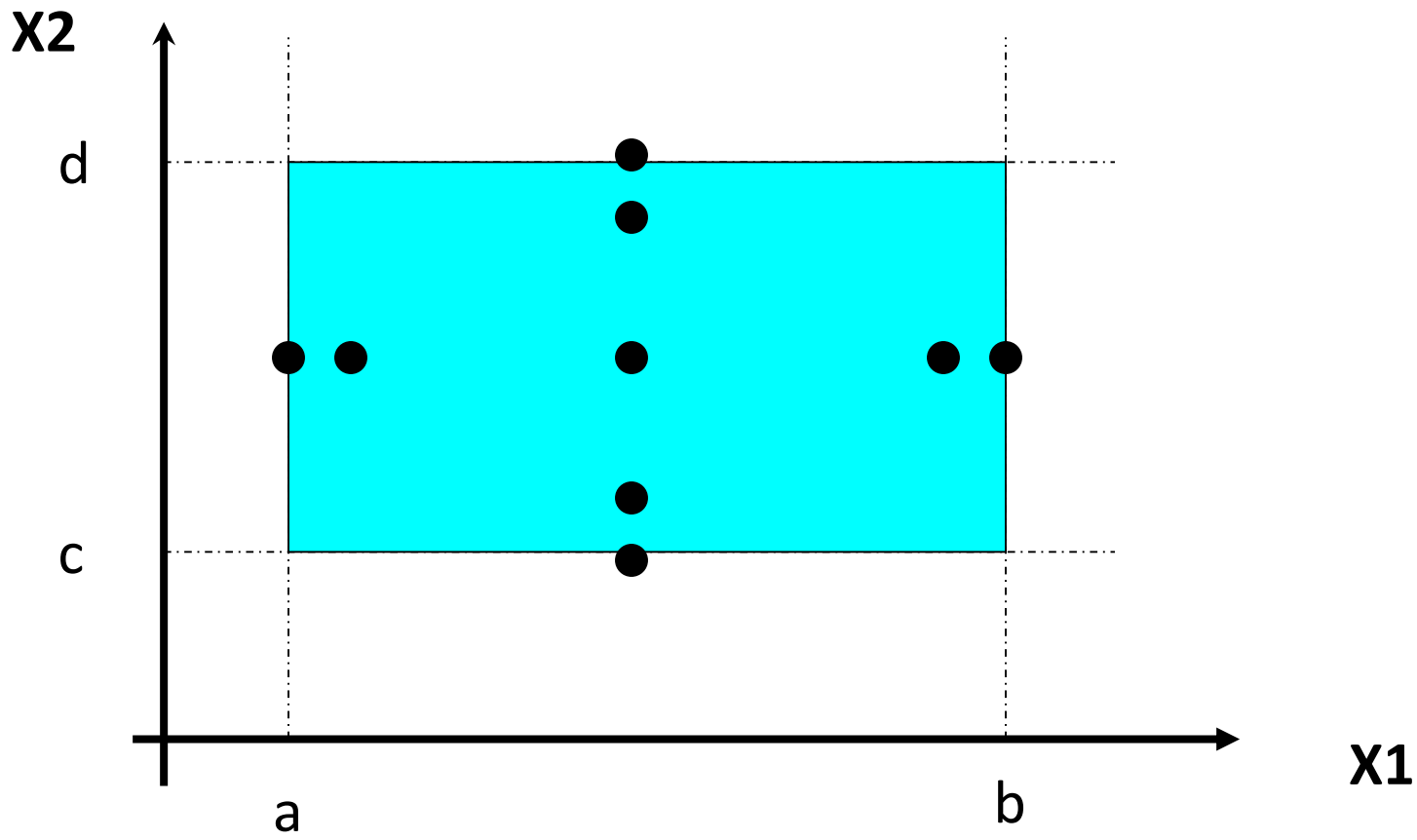
  v_i spazia tra questi 5 valori, mentre tutte le altre sono mantenute al loro valore nominale.

Dominio di 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>}

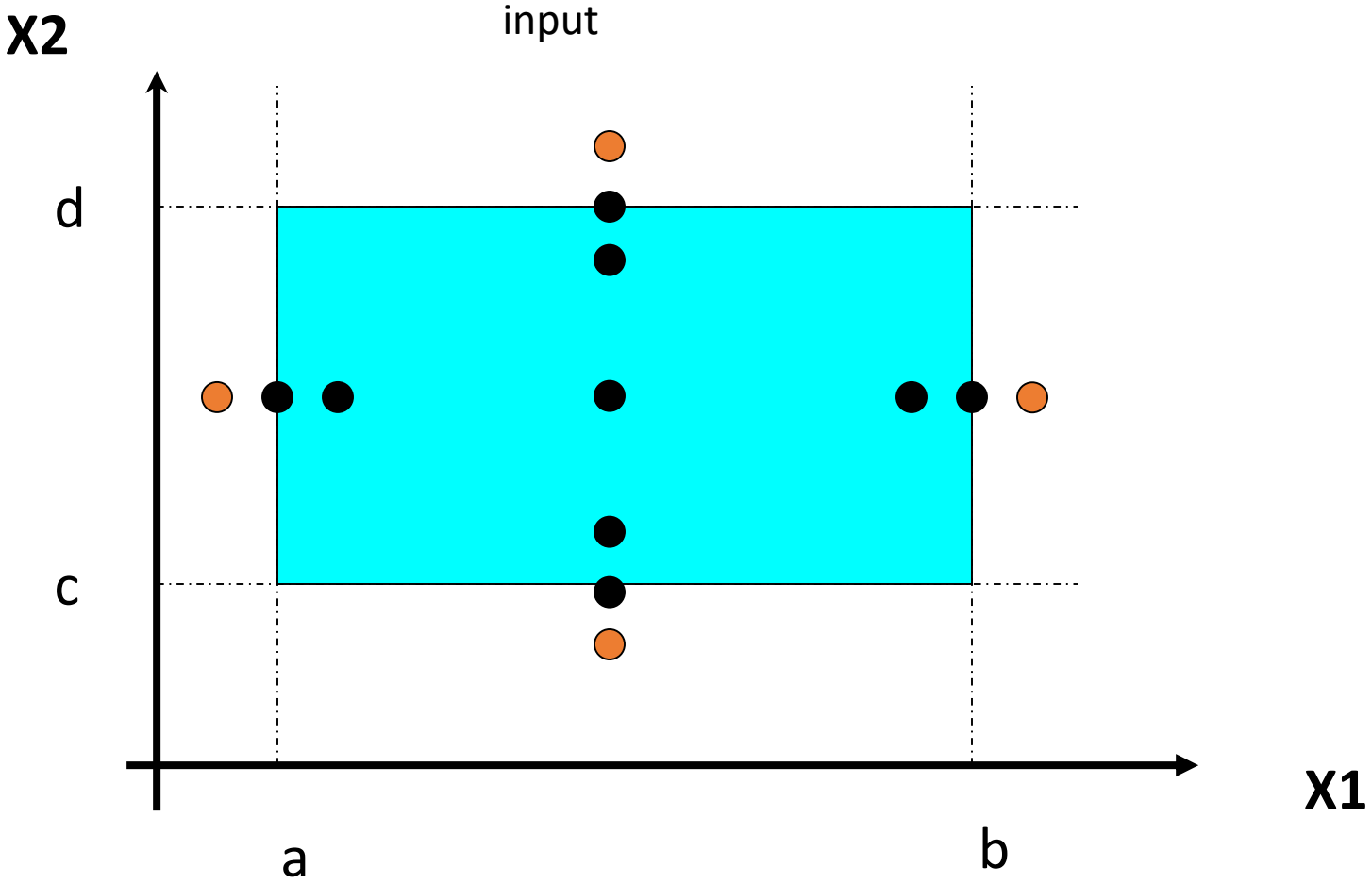


Caso Generale e Limitazioni

- 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

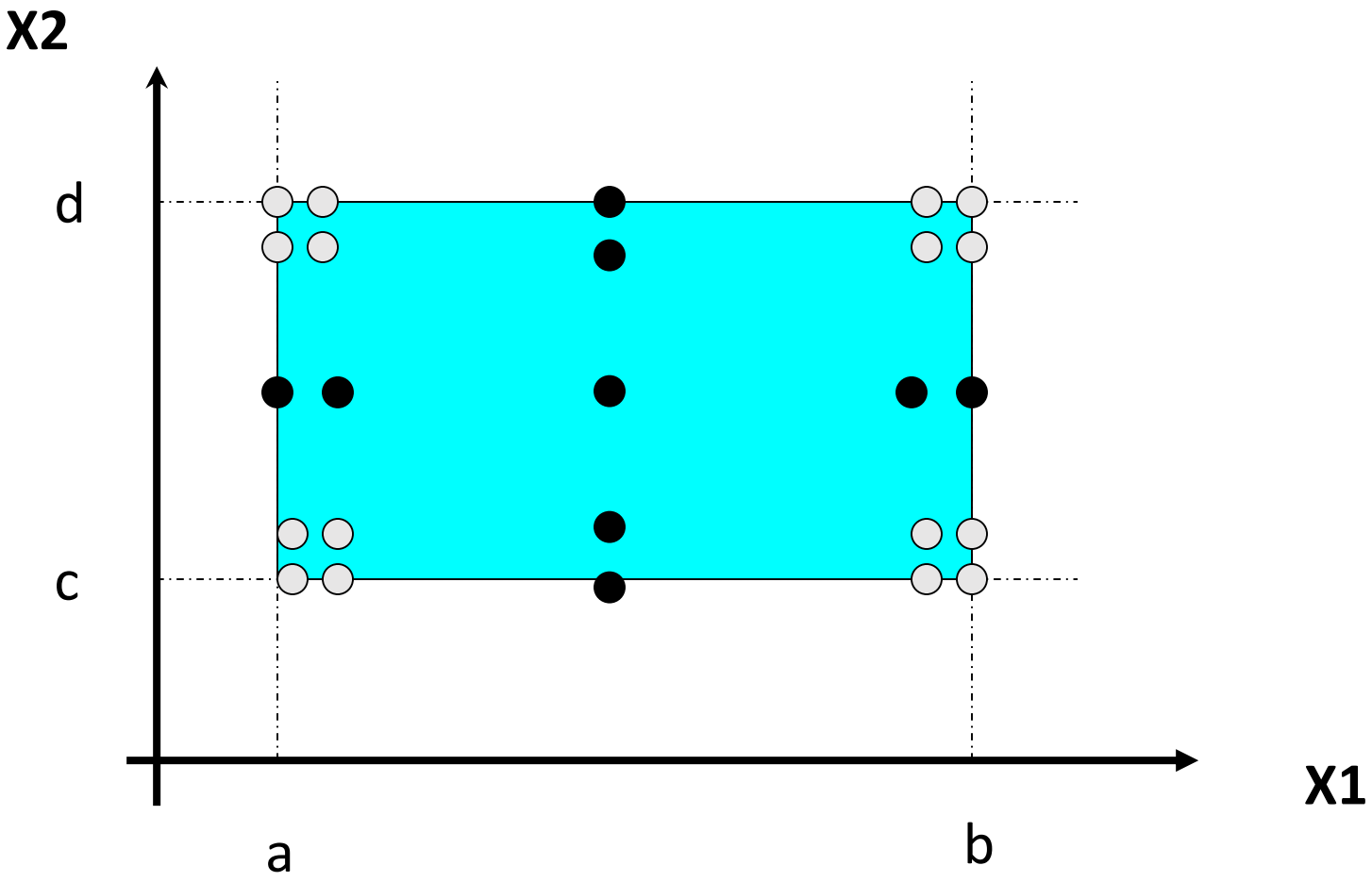
Esegue 6n+1 casi di test, date n variabili di input



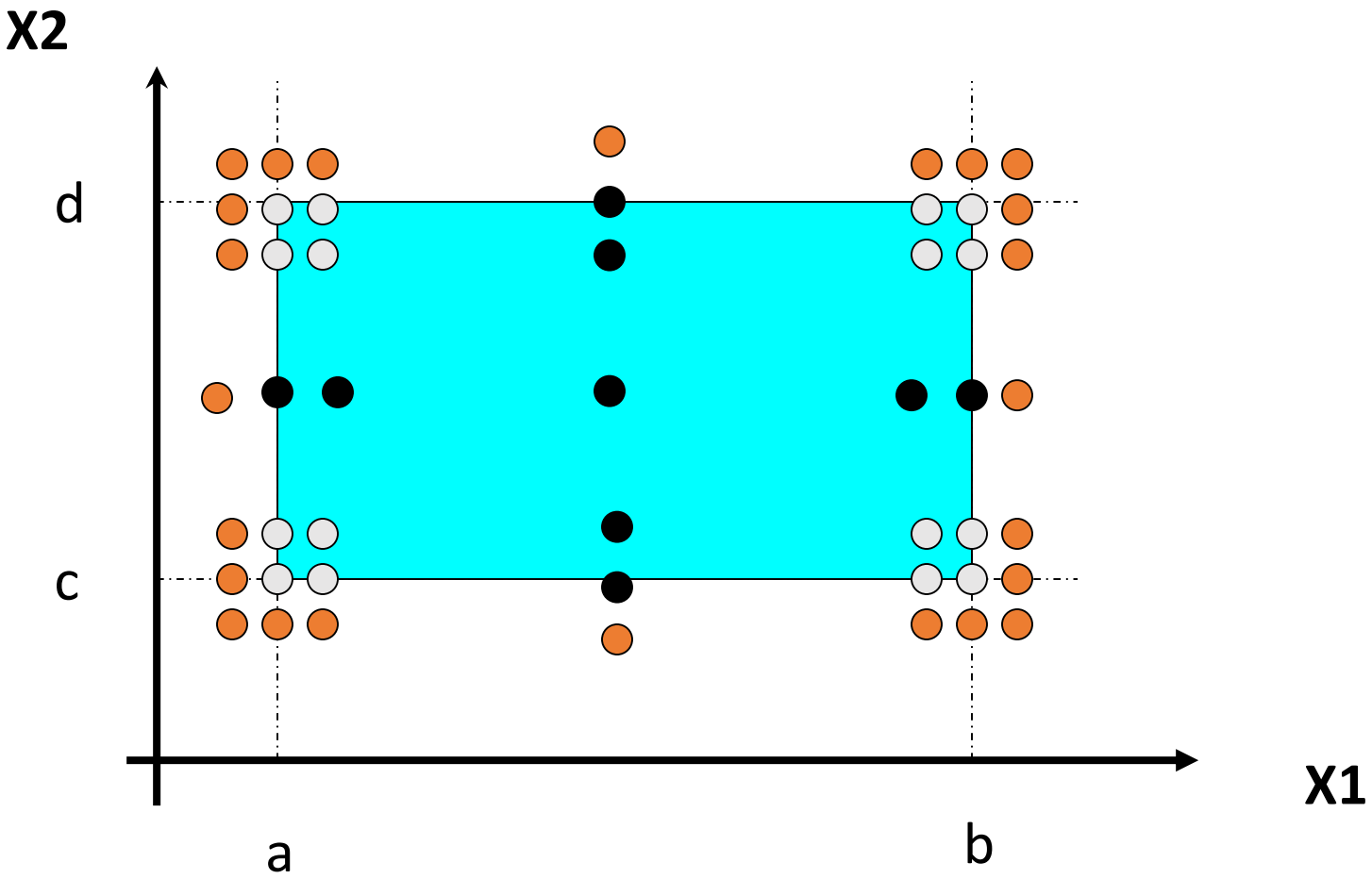
Worst Case Testing (WCT)

- 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**

Testing Strutturale (White Box)

Testing Strutturale

- La definizione dei casi di test e dell'oracolo è basata sulla struttura codice
- Si selezionano quei test che esercitano **tutte** le strutture del programma
  - 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 funzionale
- 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

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

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

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

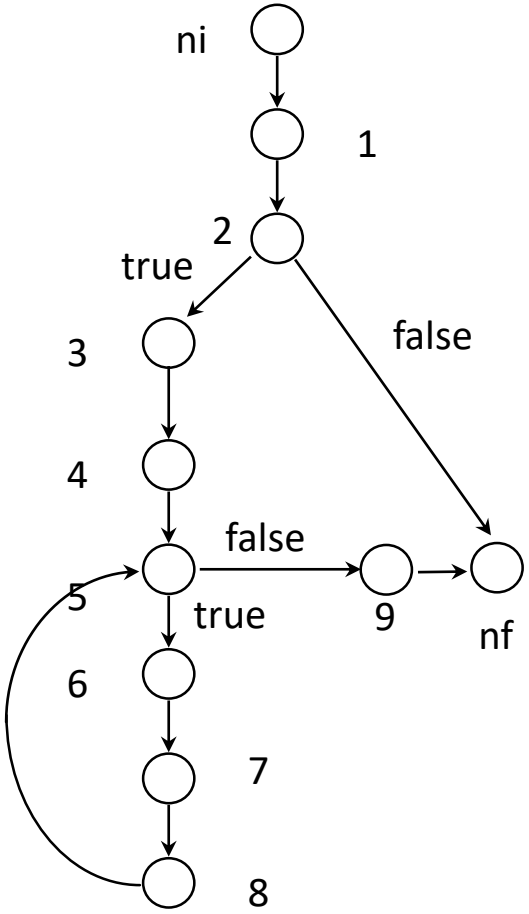
rappresenta la relazione di flusso di controllo

Un modello di rappresentazione dei programmi (2/2)

- n_i ed n_f sono detti nodo iniziale e nodo finale
- Un nodo in $\mathrm{Ns} \cup \{\mathrm{ni}\}$ 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

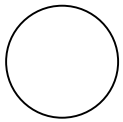
Esempio di GFC

```
void Quadrato() {
int x, y, n;
1 cin >> x;
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           cout << y;
            }
}
```



Costruzione del control flow graph per programmi strutturati

Il grafo è costruito secondo la seguente notazione:

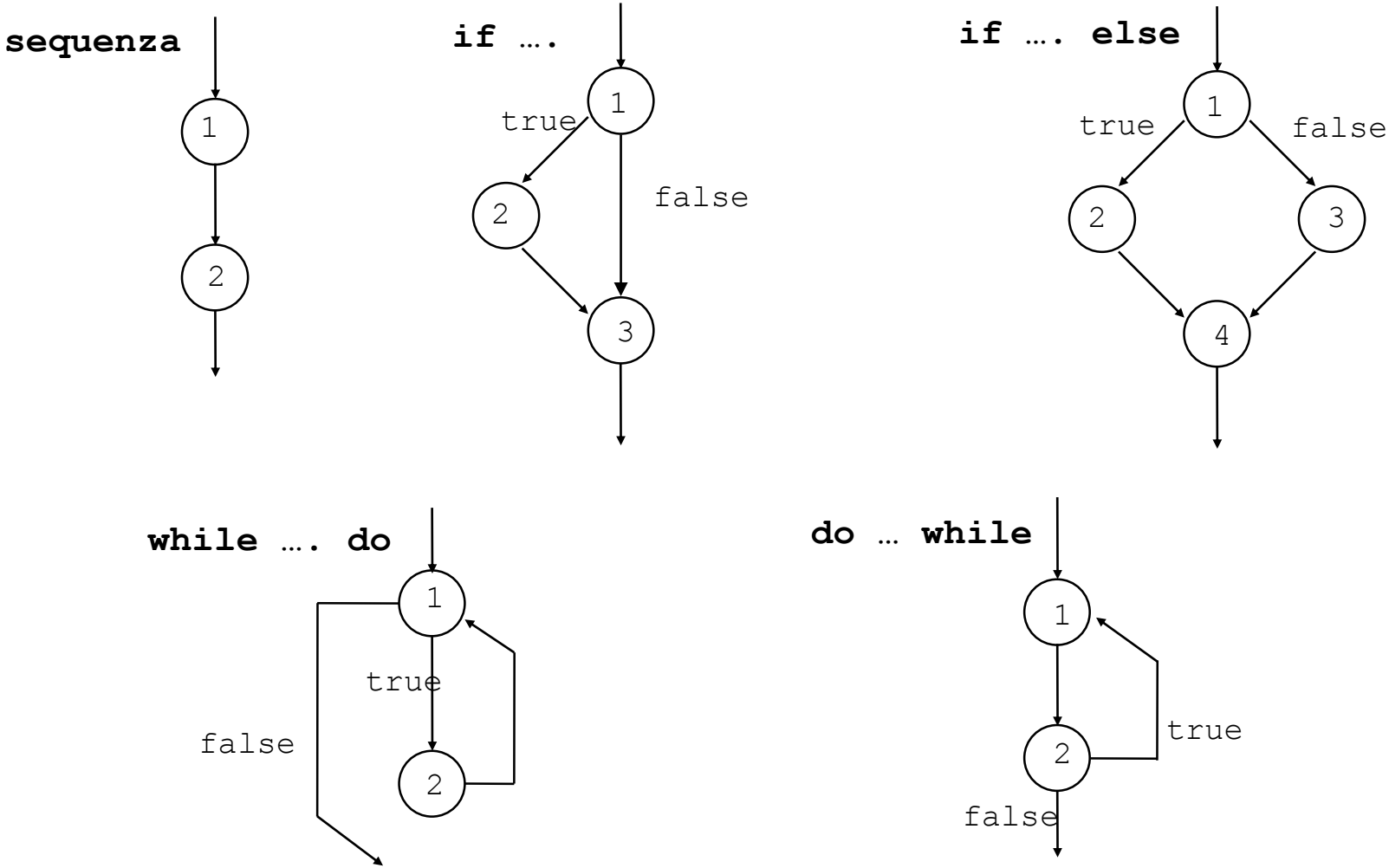


NODO rappresenta un'istruzione, identificato da un numero

ARCO rappresenta il passaggio del flusso di controllo, etichettato con {vero, falso, incond}

Un CfG può essere costruito combinando opportunamente insieme i grafi relativi alle strutture di controllo

Costruzione del control flow graph per programmi strutturati

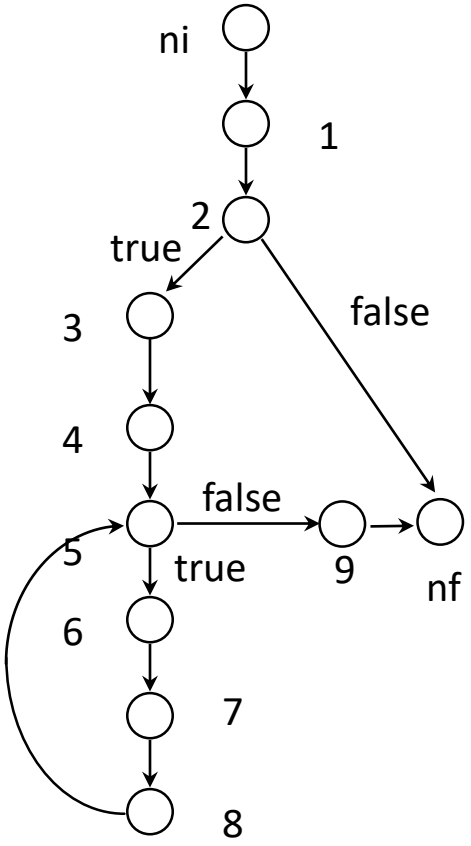


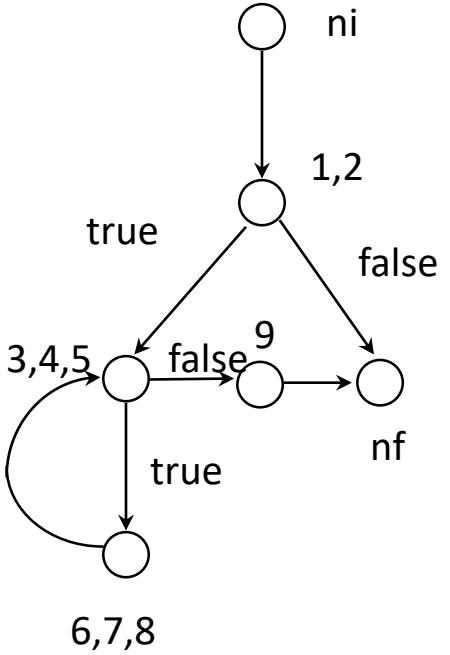
Semplificazione di un CFG

- 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)

```
void Quadrato() {
int x, y, n;
1 cin >> x;
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           cout << y;
            }
}
```





Tecniche di Testing Strutturale

- In generale fondate sull'adozione di criteri di copertura degli oggetti 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) =

oggetti coperti / # oggetti 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

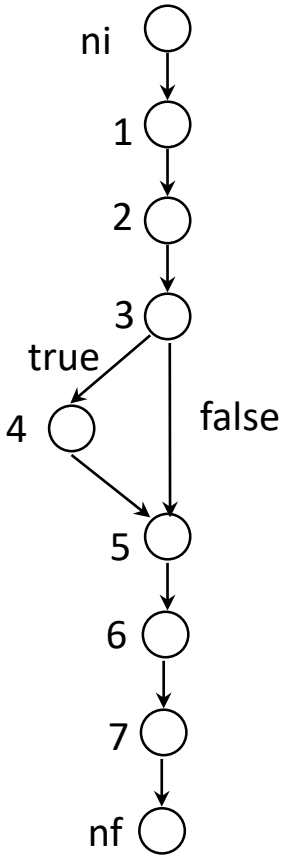
Copertura dei nodi (statement)

- 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)

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

Node Coverage: Esempio

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



Node Coverage: Esempio

- copertura del 100%: cammino 1, 2, 3, 4, 5, 6, 7
- 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)

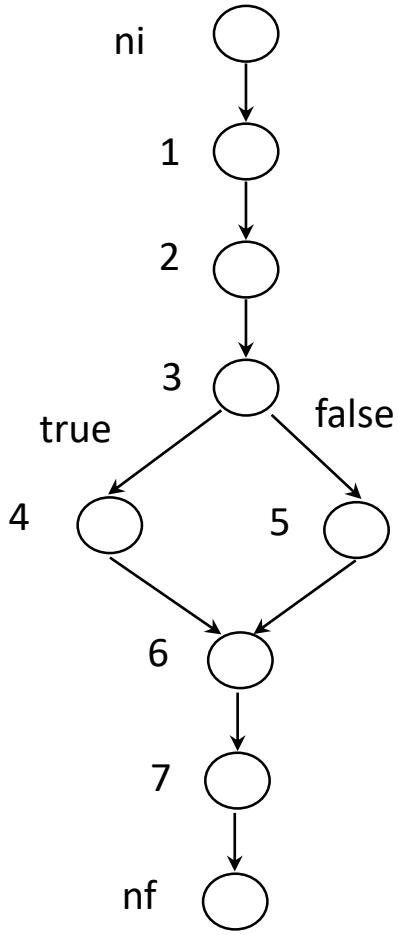
- Dato un programma P, viene definito un insieme di test case la cui esecuzione implica l'attraversamento di tutti i rami di GFC(P), ovvero l'esecuzione di tutte le decisioni di P
  - Test Effectiveness Ratio (TER)

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

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

Branch Coverage: Esempio

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



Branch Coverage: Esempio

- *cammini*:
  - a) 1, 2, 3, 4, 6, 7
  - b) 1, 2, 3, 5, 6, 7
- *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

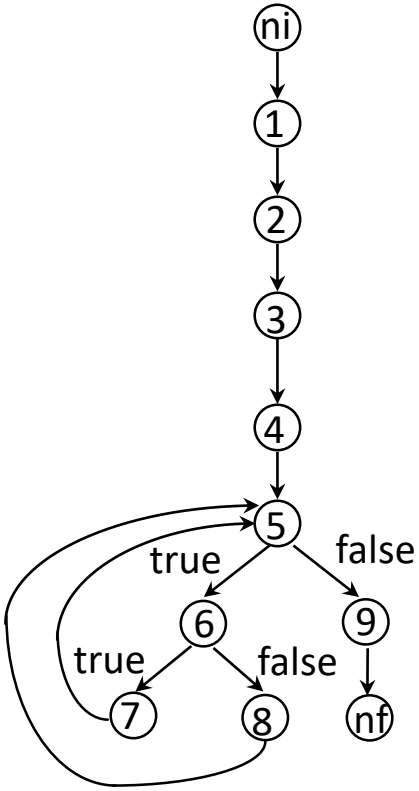
- Dato un programma P, viene definito un insieme di test case la cui esecuzione implica l'attraversamento di tutti i cammini di GFC(P)
  - Test Effectiveness Ratio (TER)

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

- 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: euristiche per limitare l'insieme dei cammini
  - Criterio di n-copertura dei cicli
  - Metodi dei cammini linearmente indipendenti (Mc Cabe)
  - …

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

Integration Testing

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:
    1. Performance Testing
    2. Pilot Testing
    3. Acceptance Testing
    4. 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.

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”*	