# Verifica e Validazione del Software

## Testing Black Box

# Riferimenti

- Ian Sommerville, Ingegneria del Software, capitoli 22-23-24 (più dettagliato sui processi)
- Pressman, Principi di Ingegneria del Software, 5° edizione, Capitoli 15-16
- Ghezzi, Jazazeri, Mandrioli, Ingegneria del Software, 2° edizione, Capitolo 6 (più dettagliato sulle tecniche)
- http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-142.pdf

# Due principali Tecniche di Testing

- Testing funzionale :
  - Richiede l'analisi degli output generati dal sistema (o da suoi componenti) in risposta ad input (test cases) definiti sulla base della sola conoscenza dei requisiti del sistema (o di suoi componenti)
    - *Spesso realizzato in modalità Black Box, ovvero senza accedere in alcun modo alla struttura interna del software*

- Testing strutturale
  - fondato sulla conoscenza della struttura del software, ed in particolare del codice, degli input associati e dell'oracolo, per la definizione dei casi di prova.
    - *Necessariamente realizzato accedendo al codice sorgente, quindi in modalità white box*

# Precisazione

- I due termini:
  - Testing Black Box
  - Testing White Box

  Non individuano da soli alcuna tecnica di testing specifica, ma solo una famiglia di tecniche di testing

- Ad esempio:
  - Testing Funzionale Black Box, Testing di unità Black Box, Testing di integrazione white box, Test di unità White Box

  Sono tecniche di testing specifiche

# Testing Black Box

- Il punto comune di tutte le tecniche «Black Box» è il fatto che il software è acceduto unicamente attraverso la sua interfaccia, senza accedere in maniera diretta al codice del componente da testare (al limite, senza accedere del tutto al codice)

- Non esiste, quindi, una sola tecnica «Black Box»!

# Testing Black Box

1. Testing basato sui requisiti
2. Testing basato sugli scenari dei casi d'uso

3. Testing con classi di equivalenza
    1. Testing con copertura minima delle classi di equivalenza
    2. Testing con copertura delle classi di equivalenza adiacenti
    3. Testing con copertura delle n-ple di classi di equivalenza
    4. Testing con copertura combinatoria delle classi di equivalenza
4. Testing con classi di equivalenza e valori limite

5. Testing a partire dalle tabelle di decisione

# 1- Testing basato sui requisiti

- **Il principio della** verificabilità **dei requisiti afferma che i requisiti dovrebbero essere testabili, cioè scritti in modo da poter progettare test che dimostrino che il requisito è stato soddisfatto.**

- **Il testing basato sui requisiti è una tecnica di convalida dove vengono progettati vari test per ogni requisito.**

# Testing basato sugli Use Case

- **Noto lo Use Case Diagram e la descrizione di tutti gli scenari dei casi d'uso**
  - **Per ogni scenario si progetta uno o più test case che lo eseguano**
- **Si eseguono manualmente o automaticamente i test case progettati**
- **La strategia di testing mira alla copertura dei casi d'uso e degli scenari**

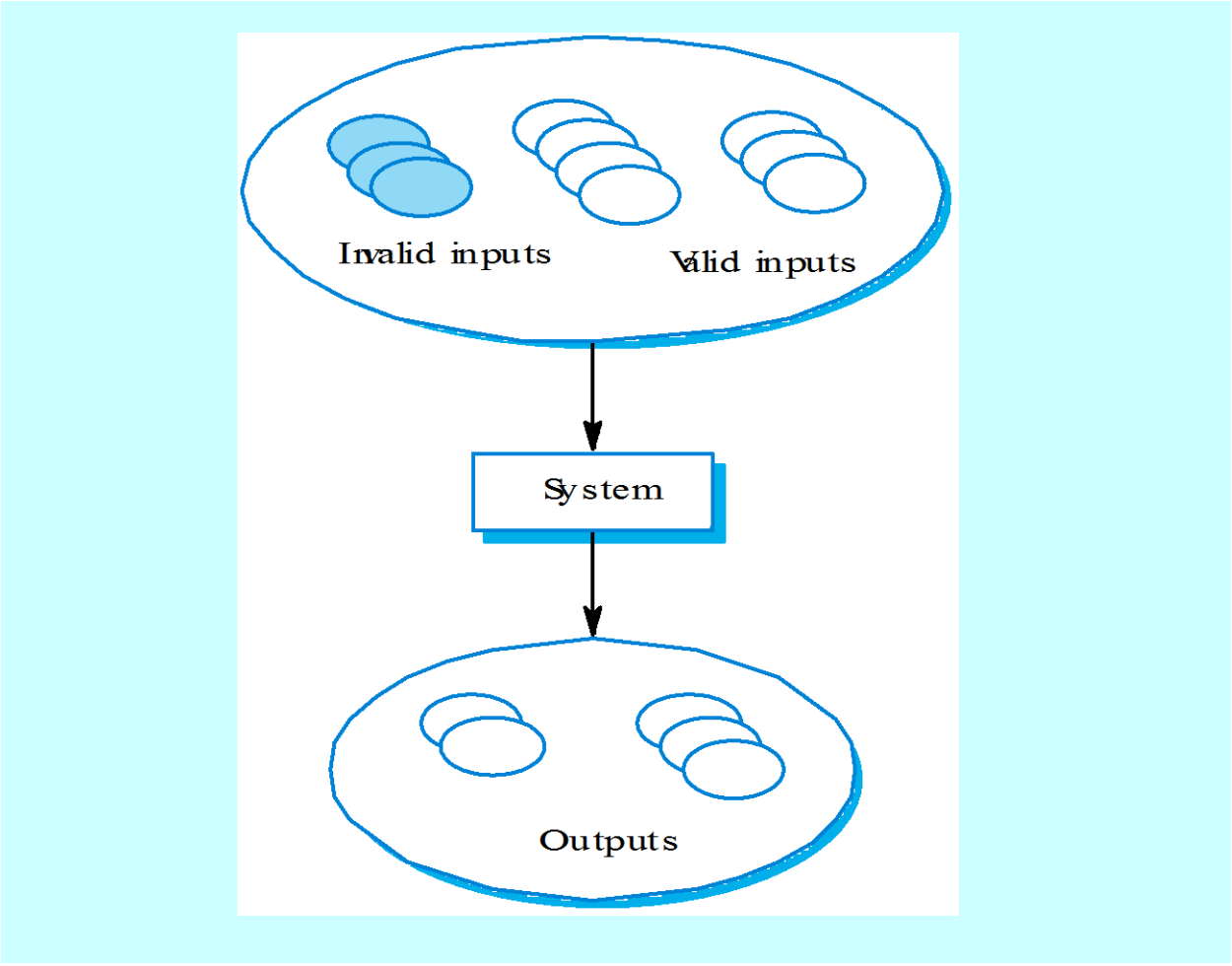
# 2- Testing delle Partizioni (o delle Classi di Equivalenza)

- **I dati di input ed output possono essere in genere suddivisi in classi dove tutti i membri di una stessa classe sono in qualche modo correlati.**

- **Ognuna delle classi costituisce una classe di equivalenza (una partizione) ed il programma si comporterà (verosimilmente) nello stesso modo per ciascun membro della classe.**

- **I casi di Test dovrebbero essere scelti all'interno di ciascuna partizione.**

# Equivalence partitioning



# Suddivisione in classi di equivalenza

- **Le partizioni sono identificate usando le specifiche del programma o altra documentazione.**

- **Una possibile suddivisione è quella in cui la classe di equivalenza rappresenta un insieme di** stati validi o non validi **per una** condizione **sulle variabili d'ingresso.**

# Ricerca delle classi di equivalenza

- Tecnica base
  - Per ogni input si ricava
    - *Una classe di equivalenza valida, corrispondente all'insieme di valori considerati validi per quell'input*
    - *Un insieme di classi di equivalenza non valide, una per ogni condizione di non validità. Ad ognuna di tali condizioni corrisponde un insieme di valori (classe d'equivalenza non valida)*

- Tecnica più dettagliata
  - Anche per le classi valide si distingue più di una classe di equivalenza, in base ai diversi scenari esercitabili

# Casi generali

- Se l'input è un:
  - intervallo di valori
    - *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*
  - valore specifico
    - *una classe valida per il valore specificato, una non valida per valori inferiori, e una non valida per valori superiori*
  - elemento di un insieme discreto
    - *una classe valida corrispondente all'insieme (tecnica classica) oppure una classe valida per ogni elemento dell'insieme (tecnica dettagliata), una non valida per un elemento non appartenente a tale insieme*
  - valore booleano
    - *Come nel caso precedente, ma per un insieme discreto a due valori (true, false)*

- In tutti i casi, è bene considerare anche un'ulteriore classe di equivalenza non valida, corrispondente alla non appartenenza dell'input al tipo atteso

# Strategie di copertura

- Copertura minima delle classi di equivalenza
  - Ogni classe di equivalenza è coperta almeno da un caso di test
    - *Numero minimo di casi di test pari al numero di classi dell'input con più classi di equivalenza*
- Copertura delle classi di equivalenza adiacenti
  - Ogni classe di equivalenza è coperta almeno da un caso di test
  - *Per ogni caso di test ne esiste almeno uno che differisce per una sola classe di equivalenza*
    - *Il numero di casi di test è nell'ordine del quantitativo totale di classi di equivalenza*
- Copertura delle n-ple di valori di classi di equivalenza (k-way)
  - *Vengono esercitate almeno una volta tutte le k-ple di classi di equivalenza diverse*
- Copertura di tutte le combinazioni di classi di equivalenza
  - Copertura di ogni combinazione di classi di equivalenza
    - *Numero di casi di test pari alla produttoria delle cardinalità dei quantitativi di classi di equivalenza di ogni classe*
    - *Equivale al caso precedente con k=numero di input*

# Esercizio

- **In un modulo bisogna inserire la propria data di nascita, composta di giorno (numerico), mese (stringa che può valere gennaio … dicembre), anno (numerico, compreso tra 1583 e 2100)**
- **Il software deve riconoscere correttamente tra date valide (corrispondenti a giorni realmente esistiti) e date non valide e fornire il giorno della settimana corrispondente per le date valide**
- **Selezionare i casi di test mediante partizionamento in classi di equivalenza**

# Le condizioni sull'input 'giorno'

- **Condizioni d'ingresso:**
  - *Il giorno può essere compreso tra 1 e 31*
- **Classi di equivalenza:**
  - ***Valida***

    **$CE_1$ : $1 \leq$ GIORNO $\leq 31$**
  - ***Non valide***

    **$CE_2$ : GIORNO $< 1$**

    **$CE_3$ : GIORNO $> 31$**

    **$CE_4$ : GIORNO non è un numero intero**

# Le condizioni sull'input 'mese'

- **Condizioni di ingresso**:
  - Il mese deve essere nell'insieme M=(gennaio, febbraio, marzo, aprile, maggio, giugno, luglio, agosto, settembre, ottobre, novembre, dicembre)

- **Classi di equivalenza**
  - **Valide**
  - $CE_5$: MESE $\in$ M

    $CE_{51}$: MESE = gennaio, $CE_{52}$: MESE = febbraio, $CE_{53}$: MESE = marzo, …. (Tot. 12 classi di equivalenza)

  - **Non valida**

    $CE_6$: MESE $\notin$ M

# Le condizioni sull'input 'anno'

- **Condizioni di ingresso**:
  - Deve essere compreso tra 1583 e 2100

- **Classi di equivalenza**
- **Valida**

  $CE_7$: 1583<= ANNO<=2100

- **Non valide**

  $CE_8$: ANNO< 1583

  $CE_9$: ANNO> 2100

  $CE_{10}$: ANNO non è un numero intero

# Selezione dei casi di test dalle classi di equivalenza

- Testing minimo con copertura delle classi di equivalenza
  - Generiamo il numero minimo di casi di test in grado di coprire ogni classe di equivalenza almeno una volta
    - *Massimizzazione dell'efficienza*

- Testing con copertura delle classi di equivalenza adiacenti
  - Generiamo casi di test che differiscano tra loro del minimo numero di classi di equivalenza coperte (idealmente una classe)
    - *Buon compromesso tra efficacia ed efficienza con conseguenze positive anche per le attività di debugging*

- Testing combinatoriale
  - Generiamo tutte le combinazioni possibili delle classi definite
    - *Massimizzazione dell'efficacia*

# Testing minimo con copertura delle classi di equivalenza

- Una Test Suite efficiente potrebbe essere la seguente:

| **Test case** | **TC1** | **TC2** | **TC3** | **TC4** |
|---|---|---|---|---|
| Giorno | 1 | 0 | 35 | primo |
| Mese | gennaio | brumaio | gennaio | gennaio |
| Anno | 1980 | 1492 | 2118 | duemila |
| **Classi coperte** | CE1, CE5, CE7 | CE2, CE6, CE8 | CE3, CE5, CE9 | CE4, CE5, CE10 |

- Tutte le classi di equivalenza sono coperte ma …
  - E' molto difficile individuare gli errori
    - Ad esempio in TC2 il sistema potrebbe rispondere con un'eccezione perchè il giorno é <1, senza valutare il mese e l'anno!

# Testing con copertura delle classi di equivalenza adiacenti

| **Test case** | **TC1** | **TC2** | **TC3** | **TC4** |
|---|---|---|---|---|
| Giorno | 1 | 1 | 1 | 1 |
| Mese | gennaio | gennaio | gennaio | gennaio |
| Anno | 1980 | 1492 | 2118 | duemila |
| **Classi coperte** | CE1, CE5, CE7 | CE1, CE5, **CE8** | CE1, CE5, **CE9** | CE1, CE5, **CE10** |

| **Test case** | **TC5** | **TC6** | **TC7** | **TC8** |
|---|---|---|---|---|
| Giorno | 1 | 0 | 35 | primo |
| Mese | brumaio | gennaio | gennaio | gennaio |
| Anno | 1980 | 1980 | 1980 | 1980 |
| **Classi coperte** | CE1, **CE6**, CE7 | **CE2**, CE5, CE7 | **CE3**, CE5, CE7 | **CE4**, CE5, CE7 |

- Tutte le classi di equivalenza valide (tecnica classica) e non valide sono coperte
  - Se un test dà esito positivo potremmo individuare subito la classe di equivalenza non valida non gestita correttamente
  - Non testiamo, però, date nel mese di febbraio, ad esempio

# Testing con copertura minima delle classi di equivalenza (classi valide elencate esaustivamente) 1/2

| **Test case** | **TC1** | **TC2** | **TC3** | **TC4** |
|---|---|---|---|---|
| Giorno | 1 | 1 | 1 | 1 |
| Mese | gennaio | febbraio | marzo | aprile |
| Anno | 1980 | 1492 | 2118 | duemila |
| **Classi coperte** | CE1, CE51, CE7 | CE1, CE52, **CE8** | CE1, CE53, **CE9** | CE1, CE54, **CE10** |

| **Test case** | **TC5** | **TC6** | **TC7** | **TC8** |
|---|---|---|---|---|
| Giorno | 1 | 0 | 35 | primo |
| Mese | brumaio | maggio | giugno | luglio |
| Anno | 1980 | 1980 | 1980 | 1980 |
| **Classi coperte** | CE1, **CE6**, CE7 | **CE2**, CE55, CE7 | **CE3**, CE56, CE7 | **CE4**, CE57, CE7 |

# Testing con copertura minima delle classi di equivalenza (classi valide elencate esaustivamente) 2/2

| **Test case** | **TC9** | **TC10** | **T11** | **TC12** |
|---|---|---|---|---|
| Giorno | 1 | 1 | 1 | 1 |
| Mese | agosto | settembre | ottobre | novembre |
| Anno | 1980 | 1980 | 1980 | 1980 |
| **Classi coperte** | CE1, CE58, CE7 | CE1, CE59, CE7 | CE1, CE510, CE7 | CE1, CE511, CE7 |

| **Test caseTC13** | |
|---|---|
| Giorno | 1 |
| Mese | dicembre |
| Anno | 1980 |
| **Classi coperte** | CE1, CE512, CE7 |

- Tutte le classi di equivalenza valide (tecnica dettagliata) sono coperte
  - Se un test dà esito positivo potremmo individuare subito la classe di equivalenza non valida non gestita correttamente
  - Non testiamo, però, date come il 30 febbraio

# Realizzazione con JUnit

- **Implementiamo in Junit i test corrispondenti a queste tre possibili test suite:**
  - **MinimaCopertura.java**
  - **CoperturaAdiacente.java**
  - **CoperturaAdiacenteDettagliata.java**

  - **Si può notare come alcuni test non siano realizzabili (il codice Junit non compilerebbe), ad esempio quelli con input testuali laddove sono richiesti numerici (es. «duemila»)**
- **Valutiamo quanti casi di test trovano problemi**

# Testing combinatoriale

- Generiamo tutte le combinazioni possibili delle classi definite
  - Ad esempio, se ci sono tre input, con rispettivamente 2, 5 e 3 classi di equivalenza, allora ci potranno essere al più 2*3*5=30 casi di test
    - *Alcuni casi di test potranno essere non eseguibili, se un valore di un campo di input impedisce l'inserimento di un altro*
      - **Ad esempio, il valore «Italia» su di un campo di input «Nazione» può causare la disabilitazione del campo di input «State» che è attivo solo se «Nazione» vale «USA»**
- Tecnica che privilegia l'efficacia all'efficienza
- La tecnica è utile specialmente in presenza di tecniche di Testing Automation

# Applicazione della tecnica combinatoria

- Nell'esempio precedente, i casi di test da generare sono 4*2*4=32 (tecnica classica) oppure 4*13*4=208 (tecnica dettagliata)
  - E non copriamo ancora il 30 febbraio
- Se consideriamo anche l'input *giorno* come discreto, i casi di test diventano 34*13*4=1768
  - Ma copriamo anche il 30 febbraio, ma non il 29 febbraio degli anni bisestili/non bisestili!
- Se considerassimo come discreto anche l'anno (2100-1582+1=519 valori possibili), i casi di test diventano 34*13*519=229398 casi di test!
  - Copriamo però tutti i casi possibili!

# Generazione automatica di test combinatori


- Uno strumento free per la generazione di test combinatori è Tobias, dell'Università di Grenoble
  - http://tobias.liglab.fr/
- Per utilizzare Tobias è sufficiente scrivere codice di test nel quale al posto dei valori è posto, tra parentesi quadre, l'elenco di valori possibili
  - Inviando un file così formattato a Tobias, esso genererà test che coprono tutte le combinazioni e invierà il file di test via posta elettronica (in formato Junit)

# Implementazione


- Tramite Tobias implementiamo casi di test Junit corrispondenti a copertura delle classi di equivalenza
  - tobiasCombinatorio.java
  - tobiasCombinatorioDettagliato.java

  - Da notare che Tobias non può generare gli oracoli (a meno che non siano costanti), quindi dobbiamo editare i test aggiungendo gli oracoli
- Valutiamo quanti casi di test trovano problemi

Testing Black Box

# Esempio

```
group EsempioBoundary[us=true] {
String
s=Calendario.calend([0,1,28,29,30,31,32],["gennaio","febbraio",
"brumaio"],[1582,1583,2100,2101]);}
```

**Codice generato (84 casi di test):**

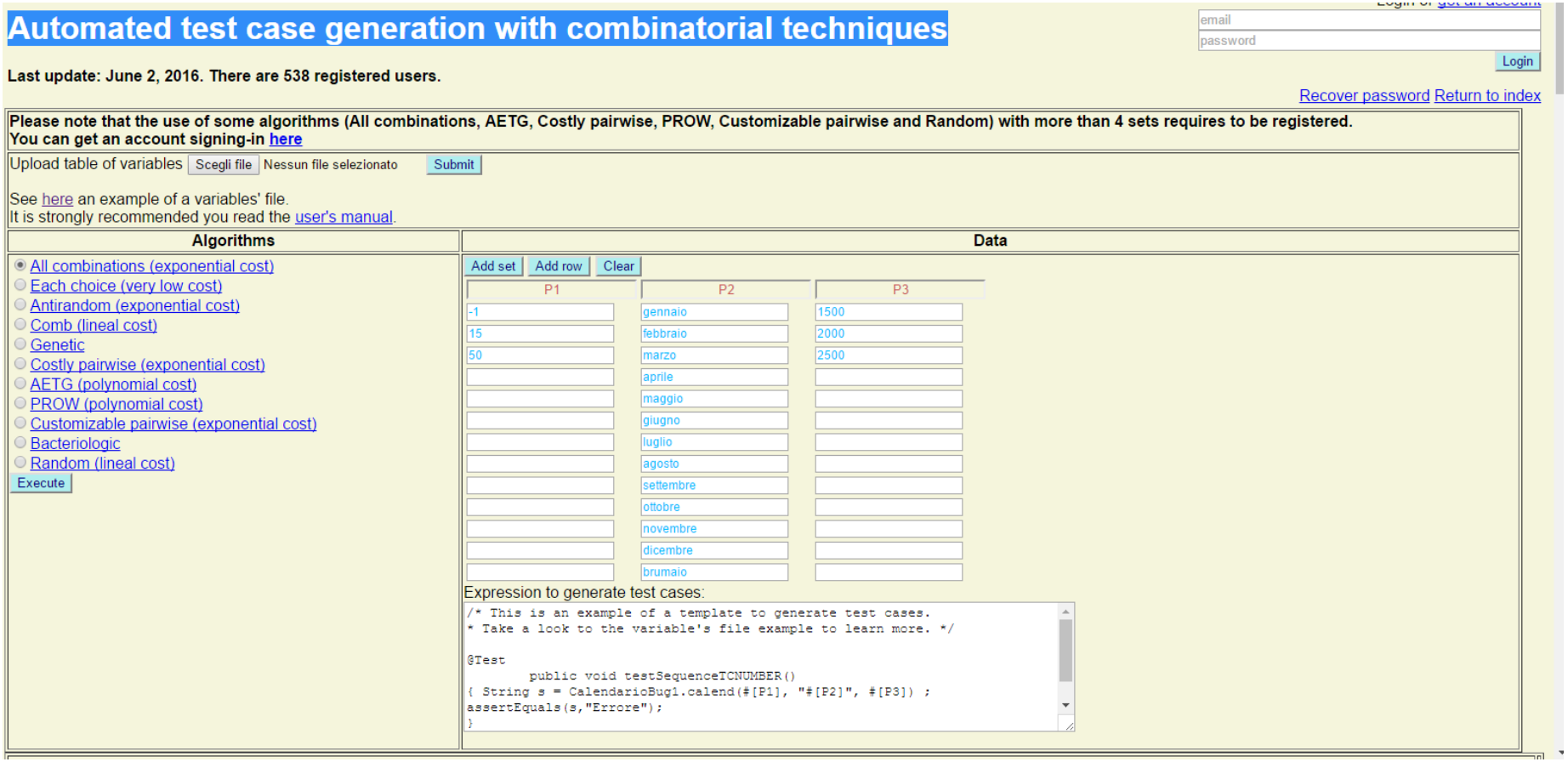
```
public class TS_Combinatorio
 {
 @Test
 public void testEsempioBoundary_1()
{ String s = Calendario.calend(0,"gennaio",1582) ;  }

 @Test
 public void testEsempioBoundary_2()
{ String s = Calendario.calend(0,"gennaio",1583) ;  }

 …
```

# Altro strumento:
# Automated test case generation with combinatorial techniques (Università de La Mancha)

Strumento web based, di molto semplice utilizzo: http://alarcosj.esi.uclm.es/CombTestWeb/combinatorial.jsp



# Output generato

## Casi di test coerenti con il formato imposto

```
/* This is an example of a template to generate test cases.
* Take a look to the variable's file example to learn more. */

@Test
        public void testSequence1()
{ String s = CalendarioBug1.calend(-1, "gennaio", 1500) ; assertEquals(s,"Errore");
}


/* This is an example of a template to generate test cases.
* Take a look to the variable's file example to learn more. */

@Test
        public void testSequence2()
{ String s = CalendarioBug1.calend(-1, "agosto", 2000) ; assertEquals(s,"Errore");
}
```

# Testing Combinatoriale

- La generazione automatica di test combinatori non comprende la definizione di oracoli
  - Dovrebbero essere aggiunti a mano!
  - E utilizzato per trovare
    - *crash*
    - *occorrenza di situazioni invarianti di fallimento (ad esempio violazioni della sicurezza, della privacy, eccessivo uso di memoria, etc.)*

- Testing Combinatoriale su NIST
  - Practical Combinatorial Testing: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-142.pdf

# Altre possibili combinazioni

- Testing combinatoriale *2-way*
  - Vengono esercitate almeno una volta tutte le coppie di classi di equivalenza diverse, ma non tutte le triple, quadruple, etc.
  - Il numero di test generati è pari al prodotto delle cardinalità dei due input aventi più classi di equivalenza
- Testing *k-way*
  - Esercita tutte le k-ple. Il numero di test generati è pari al prodotto delle cardinalità dei k input aventi più classi di equivalenza
- Il test combinatoriale puro è quindi un test *n-way*, dove n è il numero di input

- Alcuni tool che realizzano n-way testing:
  - http://www.pairwise.org/tools.asp

# Esempio testing 2-way

- Tre variabili di input, rispettivamente con 2, 3, 4 valori possibili
- Con soli 12 test (prodotto delle due cardinalità maggiori: 3*4) copriamo tutte le coppie di valori
  - (T,1), (T,2), (T,3), (F,1), (F,2), (F,3), (1,a), (1,b), (1,c), (1,d), (2,a), (2,b), (2,c), (2,d), (3,a), (3,b), (3,c), (3,d), (T,a), (T,b), (T,c), (T,d), (F,a), (F,b), (F,c), (F,d)

| Parameter name | Value 1 | Value 2 | Value 3 | Value 4 |
|---|---|---|---|---|
| Enabled | True | False | * | * |
| Choice type | 1 | 2 | 3 | * |
| Category | a | b | c | d |

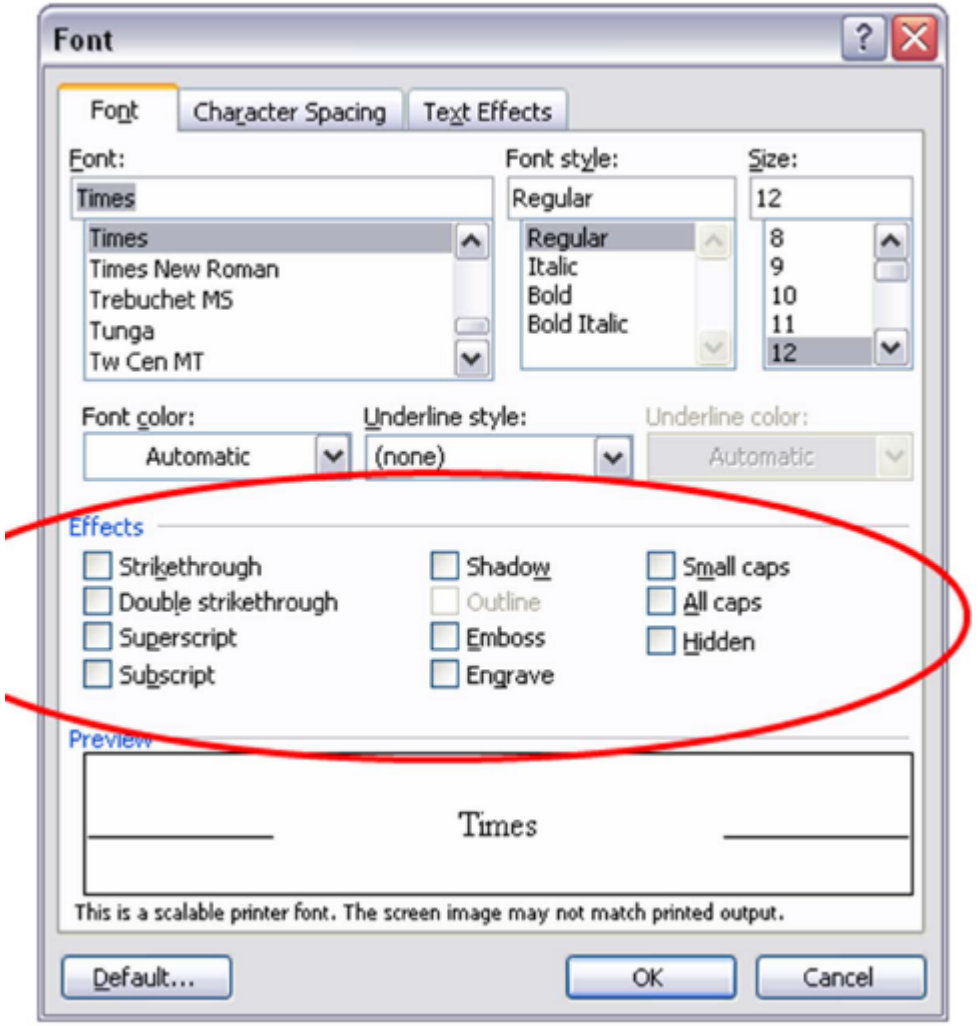
| Enabled | Choice type | Category |
|---|---|---|
| True | 3 | a |
| True | 1 | d |
| False | 1 | c |
| False | 2 | d |
| True | 2 | c |
| False | 2 | a |
| False | 1 | a |
| False | 3 | b |
| True | 2 | b |
| True | 3 | d |
| False | 3 | c |
| True | 1 | b |

# Esempi più complessi

- Quanti test combinatori?

$2^{10}$=1024 test

- 2 test per coprire ogni valore
- 10 test per coprire gli adiacenti
- 320 test 3-way



# Ordering Pizza

NIST National Institute of Standards and Technology

**Step 1** *Select your favorite size and pizza crust.*

Large Original Crust

**Step 2**

*Select your favorite pizza toppings from the pull down. Whole toppings cover the entire pizza. First ½ and second ½ toppings cover half the pizza. For a regular cheese pizza, do not add toppings.*

☑ I want to add or remove toppings on this pizza -- add on whole or half pizza.

Extra Cheese Remove

Bacon Remove

Black Olives Remove

Add toppings whole pizza

Add toppings 1st half

Add toppings 2nd half

$6 \times 2^{17} \times 2^{17} \times 2^{17} \times 4 \times 3 \times 2 \times 2 \times 5 \times 2$

= WAY TOO MUCH TO TEST

Simplified pizza ordering:

$6 \times 4 \times 4 \times 4 \times 4 \times 3 \times 2 \times 2 \times 5 \times 2$

= 184,320 possibilities

**Step 3** *Select your pizza instructions.*

☑ I want to add special instructions for this pizza -- light, extra or no sauce; light or no cheese; well done bake

Regular Sauce | Normal Cheese | Normal Bake | Normal Cut

**Step 4** *Add to order.*

Quantity 1

Add To Order | Add To Order & Checkout

# Ordering Pizza Combinatorially

NIST National Institute of Standards and Technology

Simplified pizza ordering:

6x4x4x4x4x3x2x2x5x2
= 184,320 possibilities

| | |
|---|---|
| 2-way tests: | 32 |
| 3-way tests: | 150 |
| 4-way tests: | 570 |
| 5-way tests: | 2,413 |
| 6-way tests: | 8,330 |

# Efficacia ed efficienza

- Le varie tecniche abbinano livelli crescenti di efficacia a livelli decrescenti di efficienza
  - Non è facile trovare il giusto compromesso

- L'efficacia va privilegiata quando si vuole un software affidabile (ad esempio per applicazioni critiche)

- L'efficienza va privilegiata se si vuole ridurre lo sforzo allocato alle attività di testing
  - in particolare se non può essere eseguito automaticamente

# Altro esempio

- **Una condizione di validità per un input *password* è che la password sia una stringa alfanumerica di lunghezza compresa fra 6 e 10 caratteri.**

- **Una classe** valida **CV1 è quella composta dalle stringhe di lunghezza fra 6 e 10 caratteri.**

- **Due classi** non valide **sono:**
- **CNV2 che include le stringhe di lunghezza <6**
- **CNV3 che include le stringhe di lunghezza >10**

# Problemi

- A volte non é possibile determinare staticamente le classi di equivalenza. Esempio: un sistema accetta password di tipo stringa. Classi di equivalenza possono essere:
  - Classi valide:
    - *CE1: PASSWORD corrispondente ad un utente che ha diritto d'accesso*
  - Classi non valide:
    - *CE2: PASSWORD corrispondente ad un utente che non ha diritto d'accesso*
    - *CE3: PASSWORD vuota*
- Nella descrizione dei casi di test bisogna quindi tener conto di precondizioni:

| ***Precondizione*** | ***Input*** | ***Output Atteso*** |
|---|---|---|
| 'pippo' ha diritto d'accesso | pippo | 'Accesso consentito' |
| 'pluto' non ha diritto d'accesso | pluto | 'Accesso non consentito'' |
| | Stringa vuota | 'Errore' |

# Problemi

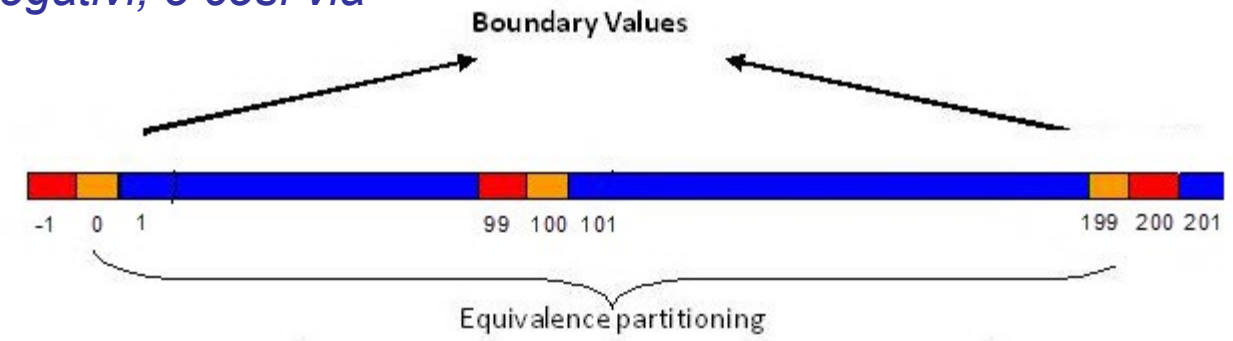
- L'appartenenza ad una classe di equivalenza dipende quindi dallo *stato* dell'applicazione
  - Non sempre è possibile determinare lo stato nè poter settare precondizioni e postcondizioni
  - In questi casi non è possibile nemmeno valutare l'efficacia del criterio, per cui l'affidabilità del test è incognita
    - *In questi casi si può solo cercare di fare quanti più test possibili, oppure ricavare i test dall'osservazione dell'utilizzo reale dell'applicazione*

# Tecnica dei valori limite (boundaries)

- Una variante alla tecnica delle classi di equivalenza consiste nel considerare anche i valori limite (boundaries)
- In pratica, vengono specializzate delle ulteriori classi di equivalenza valide e non valide corrispondenti ai valori limite degli insiemi di validità dei dati
- Si applica efficacemente a sottoinsiemi di insiemi continui  (interi, reali), in particolare ad intervalli
- Sono boundary values anche quei valori per i quali si suppone possa esserci un comportamento particolare rispetto a qualche operazione
  - Ad esempio il valore zero per un intero che potrebbe rientrare in una divisione o per un puntatore

# Casi tipici di boundaries

- Se la condizione sulle variabili d'ingresso specifica:
  - intervallo (chiuso) di valori
    - *Boundary classes: minimo dell'intervallo, massimo dell'intervallo (classi valide), valore leggermente inferiore al minimo, leggermente superiore al massimo (classi non valide)*
  - Unione di intervalli
    - *Ci sono boundary classes per ogni estremo di ogni sottointervallo*
  - Valori interi
    - *Una boundary class, indipendentemente dalle specifiche, è l'insieme {0}; un'altra, se non altrimenti considerata, è la classe dei numeri negativi, e così via*



# Esempi di boundary classes

- Per l'input giorno:
  - **{-20}: valore inferiore dell'estremo inferiore dell'intervallo**
  - {-1}: valore leggermente inferiore al valore nullo
  - {0}: valore leggermente inferiore dell'estremo inferiore dell'intervallo e anche valore nullo
  - {1}: estremo inferiore
  - **{15}: valore valido lontano dagli estremi**
  - {28}: estremo superiore in alcuni casi
  - {29}: caso critico noto
  - {30}: caso critico noto
  - {31}: caso critico noto
  - {32}: valore leggermente maggiore dell'estremo superiore
  - **{50}: valore superiore all'estremo superiore dell'intervallo**

- Per l'input anno (le specifiche del problema imponevano anno compreso tra 1900 e 2000)
  - **{1000}: valore inferiore dell'estremo inferiore dell'intervallo**
  - {1582}: valore leggermente inferiore dell'estremo inferiore dell'intervallo
  - {1583}: estremo inferiore
  - {2016}: estremo superiore
  - {2017}: valore leggermente superiore all'estremo superiore
  - **{2100}: valore superiore all'estremo superiore**

**(in grassetto i valori non boundary)**

# Esempi di boundary classes

- L'applicazione della tecnica combinatoria comporta la definizione di (7+4)*(13)*(6)=858 casi di test

– Generiamo con tobias questi casi di test (CombinatorioValoriLimite.java) e valutiamo quanti problemi hanno riscontrato

  – Riusciamo sicuramente a provare casi la validità di date come il 30 e il 31 febbraio
  – Per il 29 febbraio riusciamo a controllare la validità nei normali anni bisestili e non bisestili, ma non valutiamo il comportamento corretto in casi eccezionali come il 29 febbraio 1900 (che non è data valida)
    - *Lo avremmo valutato se avessimo saputo che 1700,1800,1900, 2100 erano valori critici per l'input anno*

# Testing basato su Tabelle di Decisione

- Le tabelle di Decisione sono uno strumento per la **specifica** black-box di componenti in cui:
  - A diverse combinazioni degli ingressi corrispondono uscite/azioni diverse;
  - Le varie combinazioni possono essere rappresentate come espressioni booleane mutuamente esclusive;
  - Il risultato non deve dipendere da precedenti input o output, né dall'ordine con cui vengono forniti gli input.

- Le Tabelle di Decisione sono primariamente una tecnica di **progettazione,** ma risultano utili anche a supporto del testing

# Costruzione della Tabella di Decisione

- Le colonne della Tabella rappresentano le combinazioni degli input a cui corrispondono le diverse azioni.
- Le righe della tabella riportano i valori delle variabili di input (nella Sezione Condizioni) e le azioni eseguibili (nella Sezione Azioni)
- Ogni distinta combinazione degli input viene chiamata Variante.

# Esercizio

- Scrivere la tabella di decisione relativa alla validità di una data che tenga conto dei seguenti vincoli:
  - Aprile, giugno, settembre, novembre hanno 30 giorni
  - Febbraio ha 28 giorni negli anni non bisestili, 29 altrimenti
  - Sono bisestili tutti gli anni divisibili per 4 e non divisibili per 100
  - Sono bisestili tutti gli anni divisibili per 400
  - Sono validi tutti gli anni successivi al 1582

# Esempio: Validità della data del giorno

| | | Varianti | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |
| **Condizioni** | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| **Azioni** | | | | | | | | | |

# Esempio: Validità della data del giorno

| | | Varianti | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| **Condizioni** | Giorno | | | | | | | | | |
| | Mese | | | | | | | | | |
| | Anno | | | | | | | | | |
| | Bisestile | | | | | | | | | |
| **Azioni** | Valida | | | | | | | | | |

# Esempio: Validità della data del giorno

| | | Varianti | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| **Condizioni** | Giorno | [1,28] | 29 | 29 | {29,30} | 31 | 30 | 31 | Qualsiasi | Qualsiasi |
| | Mese | Qualsiasi | 2 | 2 | ≠2 | ∈{1,3,5,7,8,10,12} | 2 | ∈{2,4,6,9,11} | Qualsiasi | Qualsiasi |
| | Anno | >1582 ≤2100 | >1582 ≤2100 | >1582 ≤2100 | >1582 ≤2100 | >1582 ≤2100 | >1582 ≤2100 | >1582 ≤2100 | ≤1582 | >2100 |
| | Bisestile | Qualsiasi | Sì | No | Qualsiasi | Qualsiasi | Qualsiasi | Qualsiasi | Qualsiasi | Qualsiasi |
| **Azioni** | Valida | Sì | Sì | No | Sì | Sì | No | No | No | No |

# Varianti Esplicite ed Implicite

- Nella tabella, l'operatore logico fra le condizioni è di And;
- Nell'esempio precedente abbiamo 12 condizioni sugli input e 8 varianti significative, ma in generale esistono più combinazioni possibili.
- Quante combinazioni di condizioni sono in generale possibili?
  - Per n condizioni, $2^n$ varianti (ma non tutte sono plausibili)- sono dette **varianti implicite**.
  - Il numero di **varianti esplicite (**significative) è in genere minore!

# Traccia di tesina di approfondimento teorico/pratico

- Studio di algoritmi e strumenti a supporto del testing combinatoriale
  - Confronto di strumenti / tecniche esistenti
- Proposta/realizzazione di uno strumento di testing combinatoriale
  - Nel contesto di testing di interfaccia utente / sistema / unità /…
  - Con tecnica eventuale per la generazione dell'oracolo
  - Con eventuale utilizzo di uno strumento per la generazione delle classi di equivalenza (ad esempio dizionario dei dati)

# Appendice

# Generazione dei Test

- Nota (dalla fase di progettazione) la Tabella delle Decisioni possibili strategie per la generazione dei casi di test:
  - Test suite che copre di tutte le varianti esplicite
  - Test Suite che copre tutte le varianti implicite

- Puntualizzando:
  - Le tabelle di decisione sono primariamente una tecnica di progettazione di dettaglio
  - Partendo dalle tabelle di decisione è possibile generare automaticamente casi di test
    - *E' possibile generare i casi di test anche prima di implementare la soluzione*
  - A partire dall'idea delle tabelle di decisione è stata sviluppata la teoria delle Fitness Table ed in particolare Fitnesse

# Fitnesse


- Fitnesse è un software organizzato in forma di wiki, che fornisce due funzionalità di base:
  - Scrivere test di accettazione (o di unità) in maniera collaborativa
    - *Tramite tabelle di decisione*
    - *Funziona per programmi java*
  - Eseguire test in maniera automatica

- Fitnesse si può scaricare come standalone da:

  http://www.fitnesse.org/FitNesseDownload

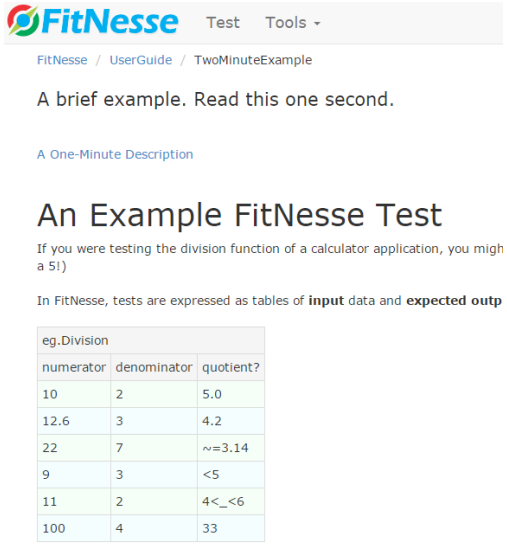
  E può essere eseguito come Web server digitando, ad esempio:

  **java -jar fitnesse-standalone.jar –p 8001**

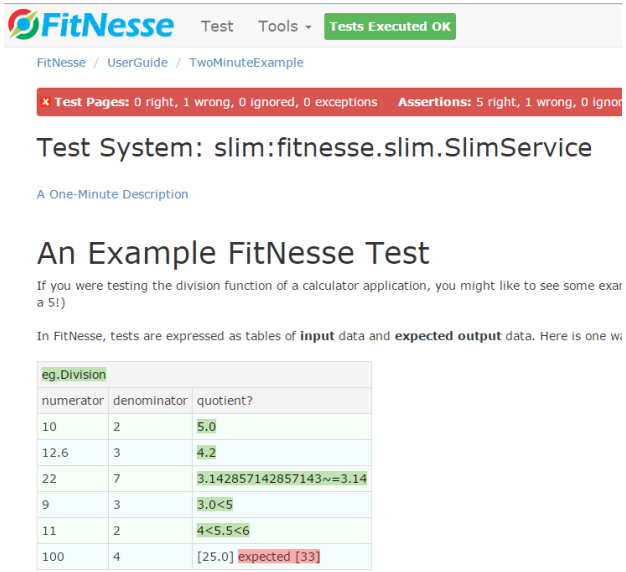
- **Fitnesse si mostra come un server di pagine wiki, che possono essere lette ed editate**
- **All'interno di pagine wiki che seguono un template di test possono essere presenti tabelle di decisione. Indicando un package e un metodo, è possibile anche eseguire i test**

# Fitnesse: esempio

- Esempi presi dalla User Guide di Fitnesse:
  - /FitNesse.UserGuide.TwoMinuteExample
    - *esempio di base*
  - /FitNesse.UserGuide.WritingAcceptanceTests.FixtureCode
    - *esempio di base ed accenno ad esempi più complessi*




- Cliccando su Test
  - Viene eseguito un metodo del package eg.Decision
  - Che prende in input due parametri denominati numerator e denominator
  - Che dovrebbe dare in output il parametro denominato quotient
  - In verde i test che hanno restituito il risultato atteso, in rosso gli altri




Il test è stato eseguito dal motore Slim che ha riportato il risultato nella stessa web page da cui è stato chiamato

# Confronto tra classi di equivalenza e tabella delle decisioni

- La tecnica di copertura della tabella delle decisioni può essere abbinata ad una tecnica di copertura delle classi di equivalenza
  - La tecnica di copertura delle tabelle di decisione si concentra nel provare tutte le combinazioni valide
  - La tecnica di copertura delle classi di equivalenza si concentra nel provare le casistiche di dati non validi
    - *In questo caso potevamo ottenere la stessa efficacia ottenuta con 100000 casi di test combinatori eseguendo non più di 20 casi di test*
      - **Ma dovevamo avere una conoscenza profonda del problema risolto dall'algoritmo!**

# Altro esempio

- Al termine del campionato di calcio di serie A del 2011, le prime due squadre si qualificano direttamente alla Champions League, mentre la terza classificata deve sottoporsi ad uno spareggio: se lo vince si qualifica per la Champions League, altrimenti per l'Europa League
- La 4° e la 5° classificata si qualificano automaticamente per l'Europa League, insieme con la squadra vincitrice della Coppa Italia, qualora essa sia arrivata 6° o peggio, altrimenti si qualifica in Europa League la 6° classificata del campionato

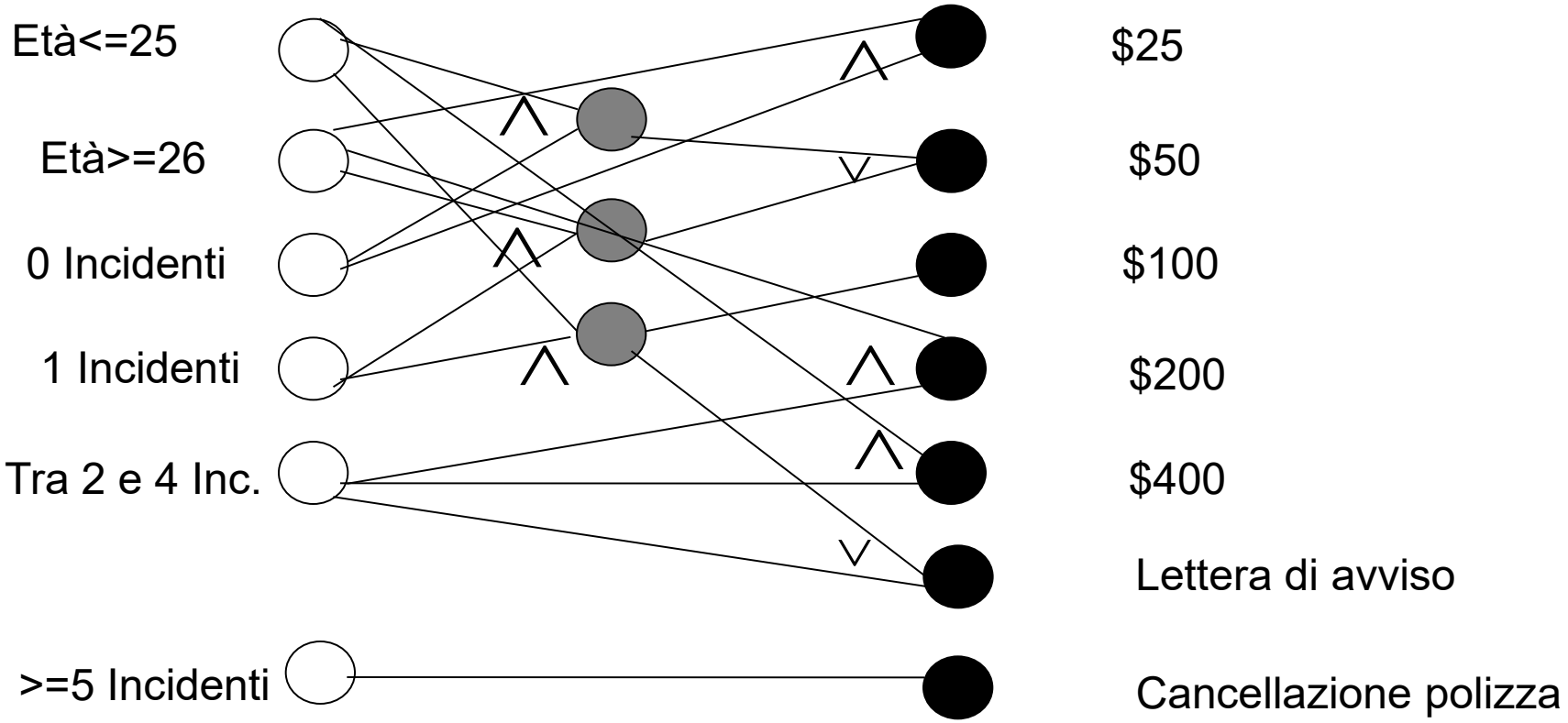
# Un esempio

| | | Varianti | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **Condizioni** | Posizione | (1°,2°) | 3° | 3° | (4°,5°) | 6° | >6° | >6° |
| | Coppa Italia | Qualsiasi | Qualsiasi | Qualsiasi | Qualsiasi | Vincitrice∈ [1°,6°] | Vinta | Non Vinta |
| | Spareggio Champions | Qualsiasi | Vinto | Perso | Qualsiasi | Qualsiasi | Qualsiasi | Qualsiasi |
| **Azioni** | Champions League | Sì | Sì | No | No | No | No | No |
| | Europa League | No | No | Sì | Sì | Sì | Sì | No |
| | Nessuna coppa | No | No | No | No | No | No | Sì |

# Testing basato su Grafi Causa-Effetto

- I Grafi Causa-Effetto sono un modo alternativo per rappresentare le relazioni fra condizioni ed azioni di una Tabella di Decisione.
- Il grafo prevede un nodo per ogni **causa** (variabile di decisione) e uno per ogni **effetto** (azione di output). Cause ed Effetti si dispongono su linee verticali opposte.
- Alcuni effetti derivano da una singola causa (e sono direttamente collegati alla relativa causa).
- Altri effetti derivano da combinazioni fra cause esprimibili mediante espressioni booleane (con operatori AND, OR e NOT).

# Il Grafo Causa-Effetto per l'esempio precedente




∧ = AND, ∨ =OR, ~= NOT

# Grafi Causa- Effetto

- Vantaggi:
  - rappresentazione grafica ed intuitiva,
  - È conveniente sviluppare tale grafo se non si ha già a disposizione una tabella di decisione
  - È possibile derivare una funzione booleana dal grafo causa-effetto (che consente di esprimere in maniera compatta tutte le possibili combinazioni di cause)
  - Può essere usata facilmente per la verifica del comportamento del software
- Svantaggi
  - al crescere della complessità della specifica, il grafo può divenire ingestibile

# Generazione dei Test

- Copertura di tutte le possibili combinazioni d'ingresso
  - Può diventare impraticabile, al crescere delle combinazioni
  - Una semplificazione: si può partire dagli effetti e percorrere il grafo all'indietro cercando alcune combinazioni degli ingressi che rendono vero l'effetto considerato.
  - Non tutte le combinazioni possibili saranno considerate, ma solo alcune che soddisfano alcune specifiche **euristiche**.
    - *Es. combinazione di OR di cause che deve essere vera -> si considera una sola causa vera per volta*
    - *AND di cause che deve essere falsa-> si considerano combinazioni con una sola causa falsa*