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

Da ISO 29119

Nella colonna di sinistra tecniche «black box»

Nella colonna centrale «white box»

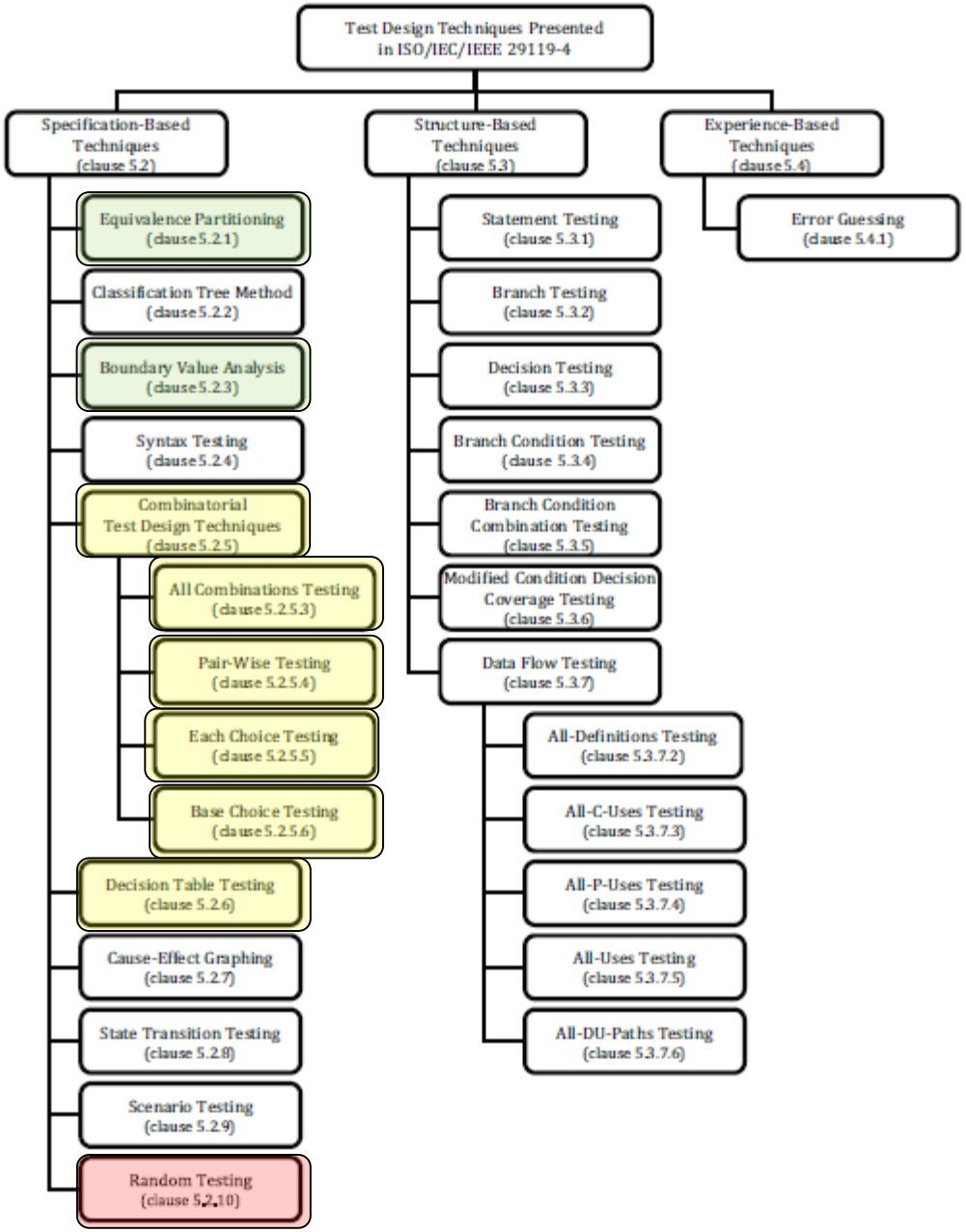


Figure 2 — The set of test design techniques presented in ISO/IEC/IEEE 29119-4

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**

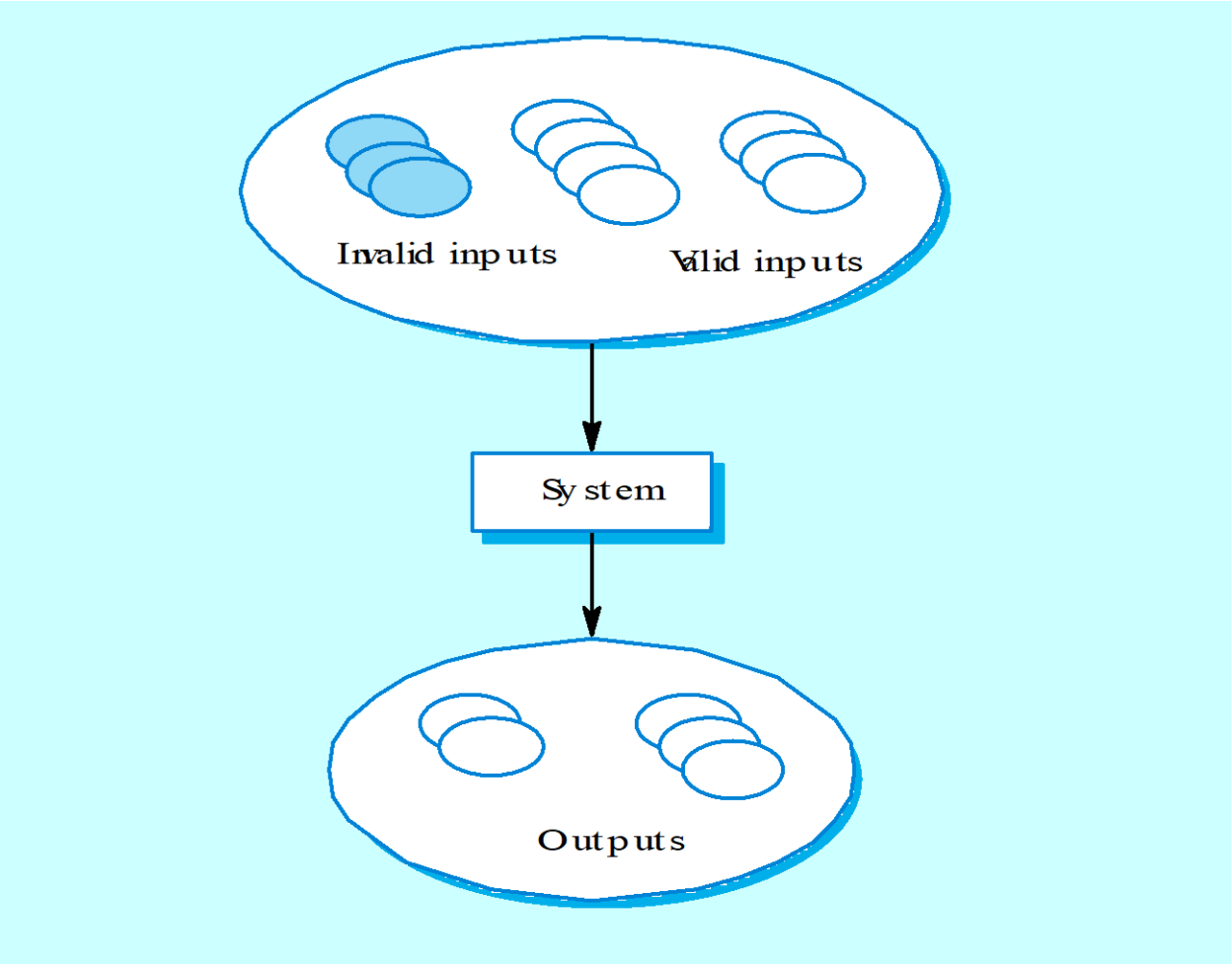
2- Testing delle Partizioni (o delle Classi di Equivalenza)

- **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



Illustrative example

- **In un metodo bisogna inserire la propria data di nascita, composta di giorno (numerico), mese (stringa), anno (numerico)**
- **Il metodo deve riconoscere correttamente tra date valide (corrispondenti a giorni realmente esistiti) e date non valide e restituire il giorno della settimana corrispondente per le date valide**

Informazioni di dominio

- **Fonte: https://it.wikipedia.org/wiki/Calendario_gregoriano**
- **In particolare:**
  - **I mesi hanno durate diverse**
  - **Il 29 febbraio esiste solo negli anni bisestili**
  - **Gli anni bisestili sono divisibili per 4, ma non per 100**
  - **Il calendario è entrato in vigore il 15 ottobre 1582**

Complessità dell'esempio

- Input:
  - Giorno – tipo intero in un intervallo limitato inferiormente e superiormente
  - Mese – stringa in un insieme di 12 valori
  - Anno – intero in un intervallo limitato inferiormente
- Output
  - Validità – booleano
  - Giorno della settimana – stringa in un insieme di 7 valori

Partition Testing

- **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.**
  - **Matematicamente, una partizione in classi di equivalenza produce insiemi disgiunti la cui unione è l'insieme totale di partenza. Inoltre la relazione di equivalenza tra partizioni gode delle proprietà riflessiva, simmetrica e transitiva**

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

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 o più classi di equivalenza valide, 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)*
      - *La suddivisione in classi valide e non valide può essere omessa, trattando tutte le classi allo stesso modo. A volte, però, la combinazione di più classi non valide risulta impossibile*

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

Classi valide e non valide

Input:

- Giorno – tipo intero in un intervallo limitato inferiormente e superiormente

  {giorno≤0}, {1≤giorno≤31}, {giorno>31}

- Mese – stringa in un insieme di 12 valori

  {mese ∈ {gennaio, febbraio, marzo, aprile, maggio, giugno, luglio, agosto, settembre, ottobre, novembre, dicembre}}, {mese ∉ {gennaio, febbraio, marzo, aprile, maggio, giugno, luglio, agosto, settembre, ottobre, novembre, dicembre}}

- Anno – intero in un intervallo limitato inferiormente

  {anno<1582}, {anno>=1582}

Classi valide e non valide

- Ulteriori classi corrispondono alla non appartenenza del valore al tipo
  - (ad esempio una stringa per giorno).
- Queste prove sono necessarie in caso di testing di sistemi senza controllo sul tipo
  - Ad esempio aventi uno stream in input, oppure moduli web, oppure protocolli
- ma non sono necessari nell'ambito di testing di unità
  - Il compilatore impedirebbe una chiamata di funzione con tipo errato

Testing strategy 1

- 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*

Omettiamo TC4 in caso di unit testing

Test case	**TC1**	**TC2**	**TC3**	**TC4**
Giorno	1	0	35	primo
Mese	gennaio	brumaio	gennaio	gennaio
Anno	1980	1492	1980	duemila

Copertura minima delle classi di equivalenza

- Le tre classi di equivalenza hanno rispettivamente cardinalità 3, 2, 2 (Omettiamo TC4 in caso di unit testing)
- Il numero di test è pari alla cardinalità massima (3)
  - Nell'ipotesi che tutti gli input siano indipendenti e sincroni
    - *Ipotesi sempre verificata in unit testing*

Test case	**TC1**	**TC2**	**TC3**	**TC4**
Giorno	1	0	35	primo
Mese	gennaio	brumaio	gennaio	gennaio
Anno	1980	1492	1980	duemila

Discussione

- E’ la tecnica che garantisce la copertura delle classi di equivalenza con il numero minimo di casi di test
  - Massima efficienza
- In caso riesca a trovare un difetto, però, può essere difficile individuarne la causa
  - Se TC1 fallisce e non TC2, il problema può essere dato da uno qualsiasi degli input o dalla loro interazione

Testing strategy 2

- 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 classi di equivalenza adiacenti

- Le tre classi di equivalenza hanno cardinalità 3, 2, 2
- Il primo test copre una classe per ogni input
- Gli altri test coprono ciascuno una sola classe non ancora coperta.

- Numero di test = somma delle classi – numero di input +1
- Esempio:
  - 3+2+2 – 3 + 1 = 5

Test case	TC1	TC2	TC3	TC4	TC5
Giorno	1	0	35	1	1
Mese	gennaio	gennaio	gennaio	brumaio	gennaio
Anno	1980	1980	1980	1980	1492

Discussione

- Tecnica meno efficiente di quella precedente
  - Il numero di casi di test cresce linearmente col numero di input
  - La tecnica è ispirata alla teoria sulla distanza di Hamming
    - *In questo caso, la test suite scelta ha distanza di Hamming pari a 1*
- Tecnica più utile ai fini del debugging
  - Se fallisce un solo test, allora ne esiste un altro che differisce per un solo valore di input che non fallisce: ci sono buone possibilità che il difetto sia stato scatenato dall'unico valore differente tra i due test
    - *Può avvenire, però, che il difetto è legato ad un'unica variabile di input ma non venga selezionata una coppia di casi di test che differisce solo per quella variabile*
      - **Ad esempio, non esiste una coppia di TC che abbiano in comune 35 e 1492**

Testing strategy 3

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

Copertura 2-way

- Per ogni 2-pla di valori, consideriamo un caso di test per ognuno dei valori della terza variabile
  - (1, gennaio) (0, gennaio) (35, gennaio) (1, brumaio) (0, brumaio) (35, brumaio)
  - (1, 1980) (0, 1980) (35, 1980) (1, 1492) (0,1492) (35, 1492)
  - (gennaio, 1980) (brumaio, 1980) (gennaio, 1492) (brumaio, 1492)

- Il numero di coppie diverse è pari a 6+6+4 = 16
  - Somma dei prodotti di coppie di cardinalità
- Ma, a causa delle sovrapposizione, sono sufficienti 6 test diversi per coprire le 16 coppie
  - Ogni test potenzialmente copre tre coppie diverse

Test case	**TC1**	**TC2**	**TC3**	**TC4**	**TC5**	**TC6**
Giorno	**1**	**0**	**35**	**1**	**0**	**35**
Mese	**gennaio**	**gennaio**	**gennaio**	**brumaio**	**brumaio**	**brumaio**
Anno	**1980**	**1492**	**1980**	**1492**	**1980**	**1492**

Altri esempi

Parameter	Values
Operating system	XP, OS X, RHL
Browser	IE, Firefox
Protocol	IPv4, IPv6
CPU	Intel, AMD
DBMS	MySQL, Sybase, Oracle

t	# Tests	% of Exhaustive
2	10	14
3	18	25
4	36	50
5	72	100

- 5 parametri, 3*2*2*2*3=72 combinazioni possibili
- Il numero di test t-way cresce esponenzialmente con t
- t=5 ➔tutte le combinazioni possibili

- http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-142.pdf

Altri esempi

- 9 parametri, 3*3*4*3*5*4*4*5*4=172800 combinazioni possibili

Parameter Name	Values	# Values
HARDKEYBOARDHIDDEN	NO, UNDEFINED, YES	3
KEYBOARDHIDDEN	NO, UNDEFINED, YES	3
KEYBOARD	12KEY, NOKEYS, QWERTY, UNDEFINED	4
NAVIGATIONHIDDEN	NO, UNDEFINED, YES	3
NAVIGATION	DPAD, NONAV, TRACKBALL, UNDEFINED, WHEEL	5
ORIENTATION	LANDSCAPE, PORTRAIT, SQUARE, UNDEFINED	4
SCREENLAYOUT_LONG	MASK, NO, UNDEFINED, YES	4
SCREENLAYOUT_SIZE	LARGE, MASK, NORMAL, SMALL, UNDEFINED	5
TOUCHSCREEN	FINGER, NOTOUCH, STYLUS, UNDEFINED	4

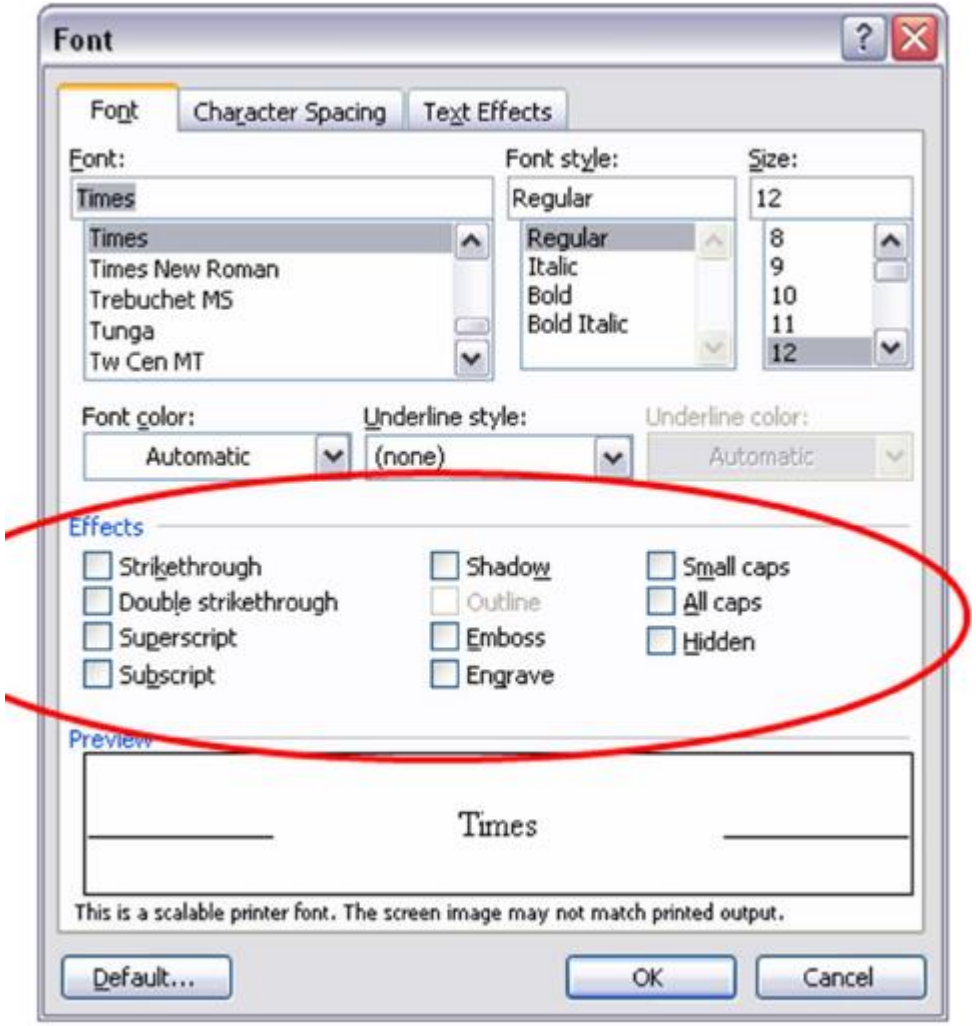
t	# Tests	% of Exhaustive
2	29	0.02
3	137	0.08
4	625	0.4
5	2532	1.5
6	9168	5.3

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

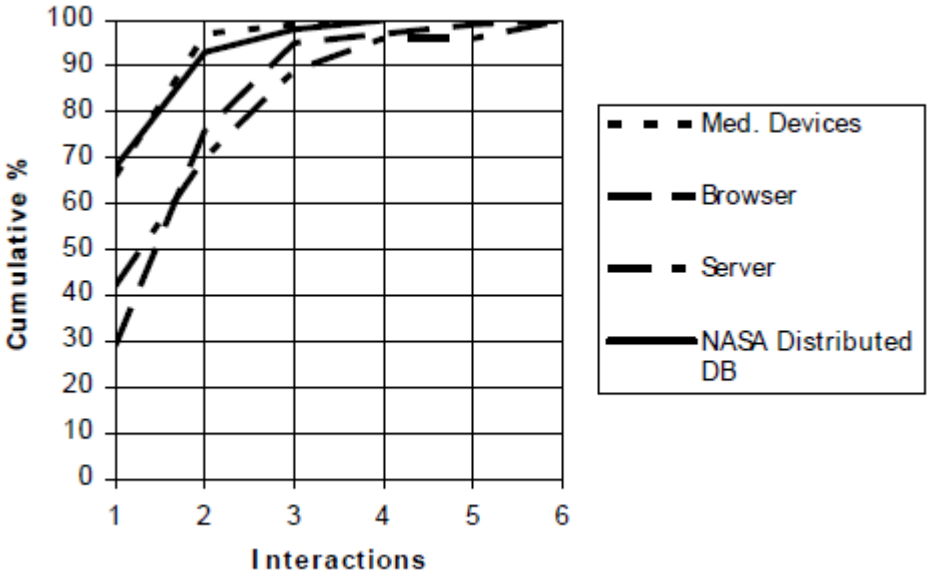
2-way tests:	32
3-way tests:	150
4-way tests:	570
5-way tests:	2,413
6-way tests:	8,330

Discussione

- Il caso 1-way è incluso con la copertura minima
  - Ogni valore è coperto da almeno un test
- Il caso n-way coincide con il test di tutte le combinazioni
- Applicare un test 2-way significa assicurarsi di aver provato tutte le combinazioni di *coppie* di valori
  - Ad esempio, se {mesi di 30 giorni, mesi di 31 giorni} e {29, 30, 31} sono due insiemi, con il test 2-way siamo sicuri di testare anche il giorno 31 di un mese di 30 giorni
  - Se, invece, vogliamo essere sicuri di testare il 29 febbraio di un anno bisestile, abbiamo bisogno di un testing 3- way

Discussione

- Il testing t-way è interpretato come il test di problemi che scaturiscono da una t-pla di valori
  - La maggior parte dei difetti dipende da singoli valori, poi da coppie, triple, etc.
  - Il numero di test, al contrario, aumenta quasi esponenzialmente con l'aumentare di t
- La scelta di t è dettata da un tradeoff tra un lento incremento del numero di difetti trovato e un veloce incremento del numero di test generati, all'aumentare di t



Finding 90% of flaws is pretty good, right?

"Relax, our engineers found 90 percent of the flaws."

http://mse.isri.cmu.edu/software-engineering/documents/faculty-publications/miranda/kuhnintroductioncombinatorialtesting.pdf

Testing Strategy 4

- Copertura di tutte le combinazioni 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 t-way con t=numero di input*

Test case	**TC1**	**TC2**	**TC3**	**TC4**	**TC5**	**TC6**
Giorno	**1**	**0**	**35**	**1**	**0**	**35**
Mese	**gennaio**	**gennaio**	**gennaio**	**brumaio**	**brumaio**	**brumaio**
Anno	**1980**	**1980**	**1980**	**1980**	**1980**	**1980**
Test case	**TC7**	**TC8**	**TC9**	**TC10**	**TC11**	**TC12**
Giorno	**1**	**0**	**35**	**1**	**0**	**35**
Mese	**gennaio**	**gennaio**	**gennaio**	**brumaio**	**brumaio**	**brumaio**
Anno	**1492**	**1492**	**1492**	**1492**	**1492**	**1492**

Discussione

- Il testing di tutte le combinazioni è il testing «esaustivo» relativamente alle classi di equivalenza
  - *Testa qualsiasi difetto che scaturisce da una specifica combinazione di classi di equivalenza*
- Il numero di test da eseguire aumenta con la produttoria del numero di classi di equivalenza
- Adatto per sistemi critici nei quali si privilegi totalmente l'efficacia nei confronti dell'efficienza

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**

Generazione automatica di test combinatori

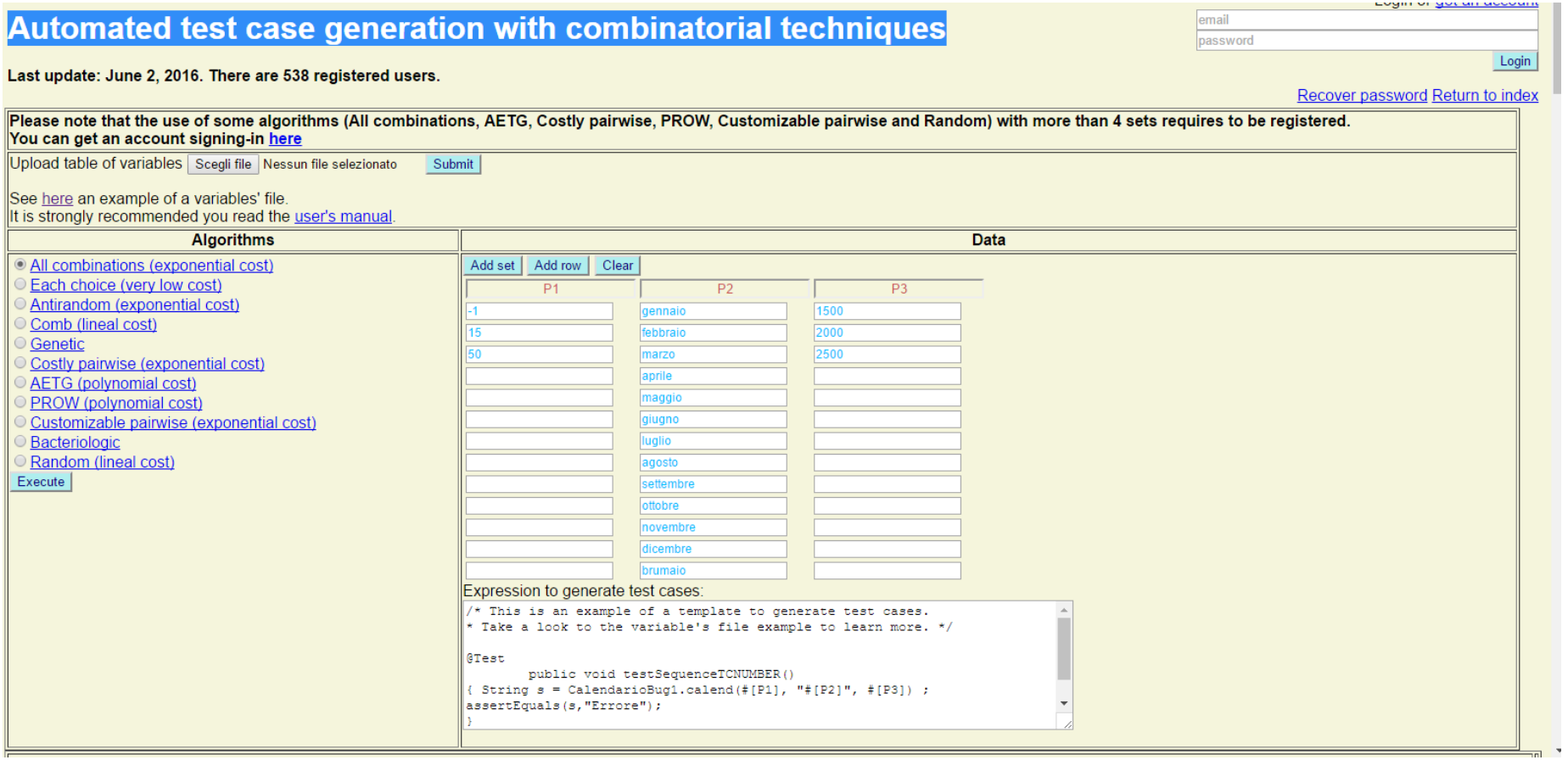
- Uno strumento free per la generazione di test combinatori è Tobias, dell'Università di Grenoble
  - http://tobias.liglab.fr/
    - *A differenza del precedente web based, non ha limitazioni sul numero di classi ma realizza solo testing all combinations*
- Per utilizzare Tobias è sufficiente scrivere 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)

Generazione automatica di test combinatori

- Uno strumento off-line (in ambiente Microsoft)
  - PICT (Pairwise Independent Combinatorial Testing tool)
    - http://download.microsoft.com/download/f/5/5/f55484df-8494-48fa-8dbd-8c6f76cc014b/pict33.msi
- Utilizzabile da linea di comando
  - Oppure con un generatore guidato in excel
    - https://osdn.net/projects/pictmaster/releases/

Automated test case generation with combinatorial techniques (Università de La Mancha)

Strumento web based, di molto semplice utilizzo:
http://alarcostest.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");
}
```

The Oracle Problem

- La generazione automatica di test combinatori non comprende la definizione di oracoli
  - Dovrebbero essere aggiunti a mano, aumentando i costi di test fino a renderlo impraticabile!

- La tecnica ritorna praticabile in alcuni specifici problemi:
  - *crash testing: l'oracolo (valutabile in maniera totalmente automatica) consiste nella valutazione della terminazione regolare del programma*
    - **In realtà ci sarebbe da considerare anche il caso di non terminazione …**
  - *occorrenza di situazioni invarianti di fallimento (ad esempio violazioni della sicurezza, della privacy, eccessivo uso di memoria, violazioni di regole di usabilità, etc.)*

The Oracle Problem

- Model Based Testing
  - La descrizione del sistema software in termini di modello consente di testare la consistenza dei comportamenti previsti dal modello con quelli riscontrati
    - *Possibilità di generare automaticamente oracoli a partire dalle specifiche*

- Regression Testing
  - L'oracolo di ogni test automaticamente generato coincide con il risultato ottenuto eseguendo lo stesso test su di un altro sistema, rispetto al quale stiamo valutando la non regressione

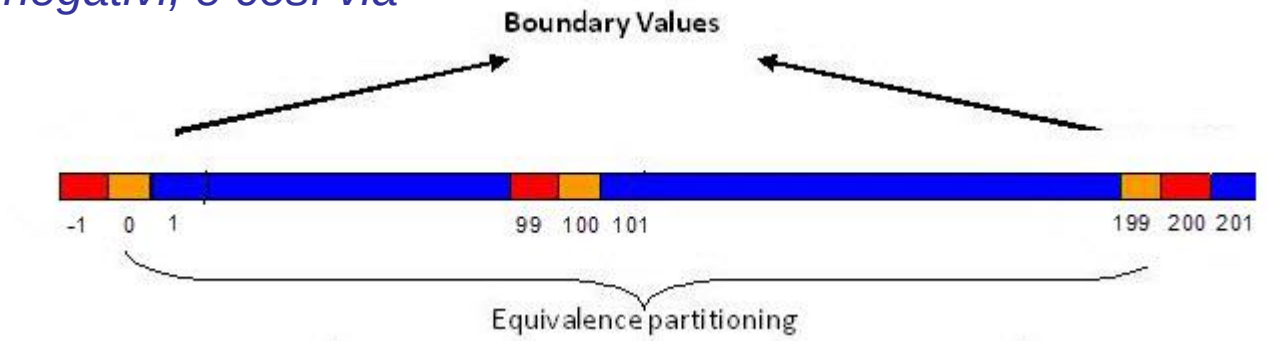
- Assertion based Testing
  - Si esprimono delle asserzioni all'interno del codice che coincidono con dei comportamenti attesi
    - *Il fallimento dell'asserzione è paragonabile ad un crash*

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:
  - **{giorno <<0}: valore inferiore dell'estremo inferiore dell'intervallo**
  - {0}: valore leggermente inferiore dell'estremo inferiore dell'intervallo e anche valore nullo
  - {1}: estremo inferiore
  - {2}: valore leggermente superiore all'estremo inferiore
  - **{giorno >>2 && giorno << 27}: valore valido lontano dagli estremi**
  - {27}: valore leggermente inferiore all'estremo superiore
  - {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
  - **{giorno >> 31}: valore superiore all'estremo superiore dell'intervallo**

- Per l'input anno
  - **{giorno << 1582}: valore inferiore dell'estremo inferiore dell'intervallo**
  - {1581}: valore leggermente inferiore dell'estremo inferiore dell'intervallo
  - {1582}: estremo inferiore
  - {1583}: valore leggermente superiore all'estremo inferiore
  - **{anno >> 1583}: valore superiore all'estremo superiore**

(in grassetto i valori non boundary)

Esempi di boundary classes

- Per l'input mese:
  - **{mese ∉ mesi dell'anno}**
  - {gennaio}
  - {febbraio}
  - {marzo}
  - {aprile}
  - {maggio}
  - {giugno}
  - {luglio}
  - {agosto}
  - {settembre}
  - {ottobre}
  - {novembre}
  - {dicembre}

In alternativa si potevano considerare le classi:

- **{mese ∉ mesi dell'anno}**
- {mese con 31 giorni}
- {mese con 30 giorni}
- {febbraio}

(in grassetto i valori non boundary)

Esempi di boundary classes

- Le tre classi hanno ora cardinalità (12, 4, 5)
  - Copertura minima: 12 casi di test
  - Copertura adiacenti: 12 + 4 + 5 – 3 + 1 = 19 casi di test
  - Copertura 2-way: 60 casi di test
  - Copertura di tutte le combinazioni: 12 * 4 * 5 = 240 casi di test
- Per coprire casi come il 31 aprile è sufficiente il 2-way testing
- Probabilmente non copriamo il 29 febbraio di un anno bisestile, perché non abbiamo specificato l'esistenza di anni bisestili

Ulteriore suddivisione

- Per l'input anno
  - **{giorno << 1582}: valore inferiore dell'estremo inferiore dell'intervallo**
  - {1581}: valore leggermente inferiore dell'estremo inferiore dell'intervallo
  - {1582}: estremo inferiore
  - {1583}: valore leggermente superiore all'estremo inferiore
  - **{anno >> 1583, divisibile per 4 ma non per 100}: anno bisestile**
  - **{anno >> 1583, divisibile per 400}: anno non bisestile**

  - **Non consideriamo la classe**
    - ***{anno >> 1583, non divisibile per 4}: anno non bisestile***

    Perché abbiamo già un rappresentante (1583)

- Ora le tre classi hanno ora cardinalità (12, 4, 6)

  - Copertura minima: 12 casi di test
  - Copertura adiacenti: 12 + 4 + 6 – 3 + 1 = 20 casi di test
  - Copertura 2-way: 72 casi di test
  - Copertura di tutte le combinazioni: 12 * 4 * 6 = 288 casi di test

- **Per coprire il 29 febbraio di un anno bisestile abbiamo bisogno di un testing 3-ways (all combinations)**

Decision tables

- 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
  - Il calendario è valido a partire dal 15 ottobre 1582

Esempio: Validità della data del giorno

		Varianti														
Condizioni	Giorno															
	Mese															
	Anno															
Azioni	Valida															

Esempio: Validità della data del giorno

		Varianti													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Condizioni	Giorno	[1,28]	29	29	{29,30}	31	30	31	Any	Any	<15	≥15	Any	<1	>31	Any
	Mese	Any	2	2	≠2	∈{1,3,5,7,8,10,12}	2	∈{2,4,6,9,11}	Any	<10	10	10	>10	Any	Any	≠ [1,12]
	Anno	>1582	>1582 Bisest	>1582 Non bisest	>1582	>1582	Any	Any	<1582	1582	1582	1582	1582	Any	Any	Any
Azioni	Valida	Sì	Sì	No	Sì	Sì	No	No	No	No	No	Si	Si	No	No	No

Varianti Esplicite ed Implicite

- Nella tabella, l'operatore logico fra le condizioni è di And;
- Nell'esempio precedente abbiamo 23 condizioni sugli input e 15 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!

Discussione

- I test possono essere generati automaticamente dalle tabelle di decisione
  - E anche gli oracoli, se abbiamo riempito la riga delle actions
  - Ci sono framework che supportano la scrittura di tabelle di decisione, la generazione, esecuzione e valutazione dei casi di test, specialmente a supporto di attività di testing di accettazione
    - *Fitnesse:* http://www.fitnesse.org/
- Le tabelle di decisione hanno la massima complessità in termini di sforzo necessario alla loro progettazione ma la massima efficacia ed efficienza
  - Tenuto conto degli ulteriori vincoli definiti, la loro efficacia è pari a quella di un testing combinatoriale con 8 * 8 * 4 = 256 casi di test

Limiti delle tecniche combinatorie

- Si tratta pur sempre di tecniche Black Box
  - Cerchiamo di coprire tutti e soli i *comportamenti* attesi a partire dalle specifiche, per l'unità sotto test
- L'estensione verso attività di testing di componenti più complessi non è banale
  - Nell'ambito di testing di sistemi interattivi, riveste importanza *l'ordine* di inserimento degli input, per cui non è più vero che tutte le combinazioni sono eseguibili
  - Nell'ambito di sistemi real time, riveste importanza l'istante di inserimento degli input. L'istante può essere un valore continuo, che dobbiamo discretizzare in un insieme di classi limitate
- All'aumentare della complessità del sistema sotto test, il testing combinatorio diventa rapidamente impraticabile, rendendo necessario il ricorso a tecniche euristiche di riduzione
  - Random Testing
  - Priority Testing
  - ...

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**

Classi di equivalenza dipendenti da precondizioni

- 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'

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

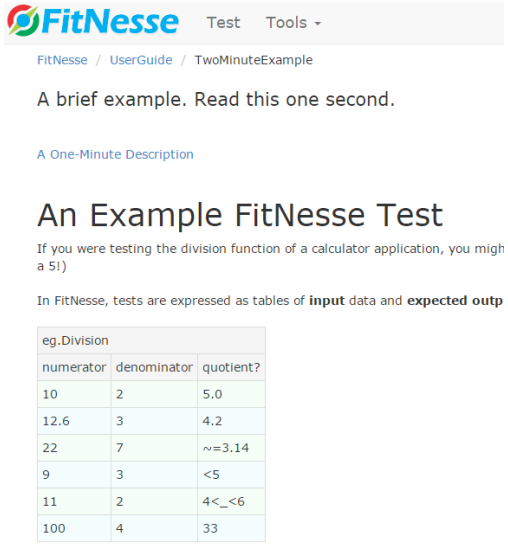
  E può essere eseguito come Web server digitando, ad esempio:

  java -jar fitnesse-standalone.jar –p 8001

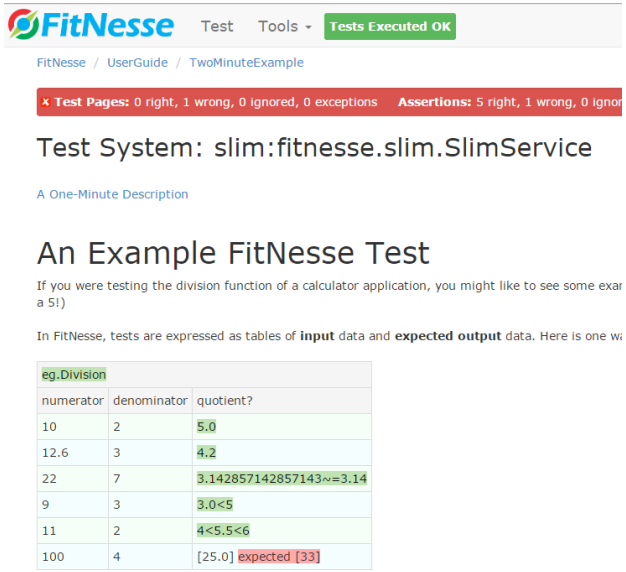
- **Fitnesse si mostra come un server di pagine wiki, che possono essere lette ed editate**
- **All'interno di pagine wiki che seguono un template di test possono essere presenti tabelle di decisione. Indicando un package e un metodo, è possibile anche eseguire i test**

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

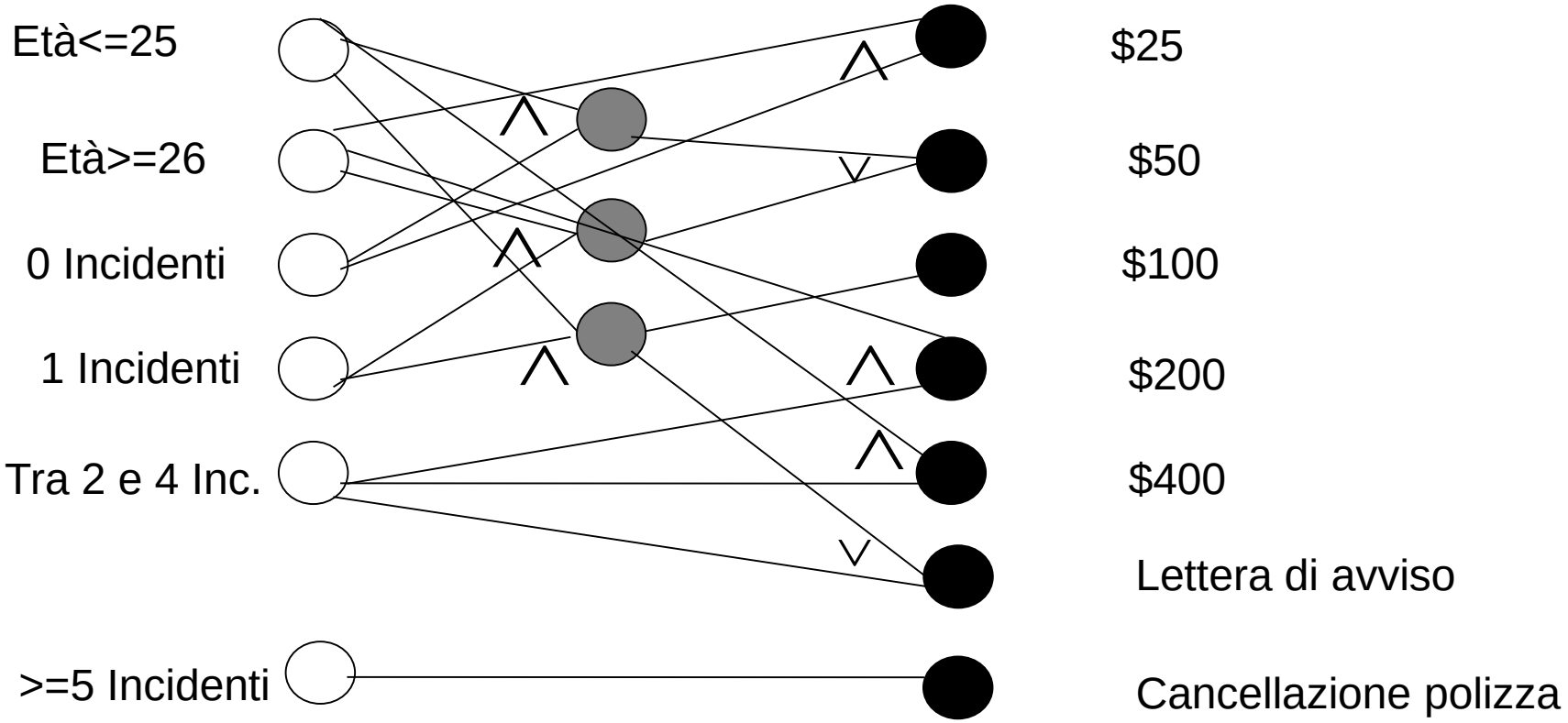
Un esempio

		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*