
Validation and Verification (Validazione e Verifica)

1

Outline

- **V and V**
 - Definizioni, terminologia
 - Tecniche
 - Storia
 - **Tecniche dinamiche**
 - Defect Testing
 - Teoria
 - Testing e fasi
 - White box e black box
 - Object Oriented Testing
 - Statistical testing
 - Debugging
 - Test Driven Development
 - Pair Programming
 - **Tecniche statiche**
 - Static analysis
 - Inspection
 - **Pianificazione della V and V**
-

2

V and V

3

V and V

Validation

- is it the *right* software system?
- esterno (verso utente finale)

Verification

- is the software system *right*?
- Interno (trasformazioni / estensioni di documenti)



4

V and V

• **E' un processo, si deve applicare a tutti i prodotti / attivita' del processo software**

• **Obbiettivi principali:**

- Scoprire difetti (fault + failure)
- Accertare se un sistema software e' affidabile in una situazione operativa

5

V& V obbiettivi

- **Dare confidenza che il software e' adeguato al bisogno**
- **Non e' possibile dimostrare che il sistema software e' defect free**
- **Si puo' assicurare che sia 'goodenough' all'uso per cui e' fatto – l'uso stesso potra' definire il livello di confidenza**

6

V & V confidenza

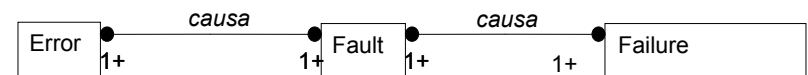
• **Dipende dall'obbiettivo del sistema, aspettative dell'utente e contesto di marketing**

- Funzionalita'
 - Il livello di confidenza dipende dal tipo di sistema
 - Safety critical, mission critical, altro
- Aspettative dell'utente
 - Possono essere basse per certi tipi di software
- Contest Marketing
 - Avere un prodotto sul mercato ora puo' essere piu' importante che avere un prodotto affidabile domani

7

Terminologia

- **Failure**
 - evidenza di malfunzionamento rispetto a attese (implicite) o specifiche (esplicite e scritte)
- **Fault**
 - Elemento di software system che causa failure
 - Puo' essere dovuto a elemento mancante (es requisito) o meno
- **Defect**
 - Failure o fault
- **Error**
 - Causa di fault (ex. Errore battitura)

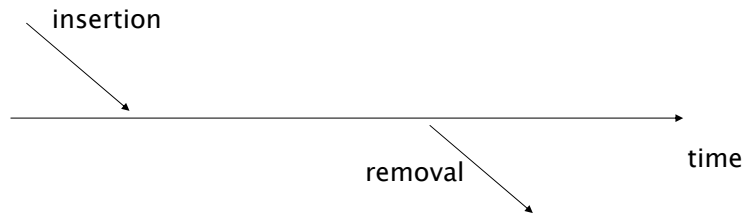


8

Insertion / removal

•Difetto ha fase/attivit 

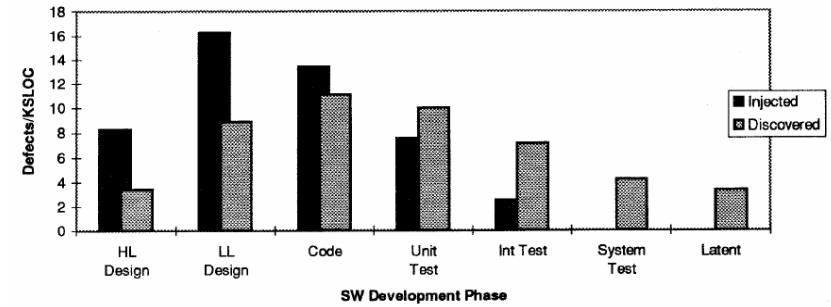
- Di inserimento
- Di removal



9

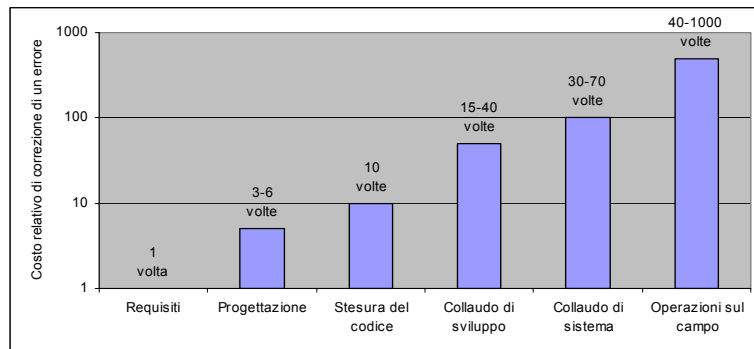
Insertion/removal by phase

Defects/KSLOC Injected or Discovered In Phase



10

Removal cost



11

Removal activities

•Each development activity should have a quality control phase + quality exit criteria



•Quality control techniques

- Static
 - Inspection, review,
- Dynamic
 - Test

12

Insertion phase	Insertion cause	Removal activity
Requisiti	Processo di raccolta dei requisiti e formulazione di specifiche funzionali	Analisi e revisione dei requisiti
Design di alto livello	Lavoro di design	Ispezioni sul design di alto livello
Design di basso livello	Lavoro di design	Ispezioni sul design di basso livello
Scrittura del codice	Codifica	Ispezioni sul codice
Integrazione/costruzione	Processo di integrazione e costruzione	Test di verifica sulla costruzione
Test di unità	Correzioni errate	Test stesso
Test di componente	Correzioni errate	Test stesso
Test di sistema	Correzioni errate	Test stesso

		Insertion phase									
		RQ	I0	I1	I2	UT	CT	ST	Operation	Total	
Removal phase	RQ	N_{11}								N_1	
	I0	*	N_{22}							N_2	
	I1	*	*	N_{33}						N_3	
	I2	*	*	*	*					*	
	UT	$N_{ij} (i \neq j)$	*	*	*	*				N_4	
	CT	*	*	*	*	*	$N_{ij} (i=j)$			*	
	ST	*	*	*	*	*	*	*		*	
	Operation	*	*	*	*	*	*	*	N_{kk}	N_k	
	Total	N_1	N_2	N_3	*	*	N_j	*	N_k	N	

Indicatori

•Phase defect efficiency

$$PDE_i = \frac{N_i}{\sum_{m=1}^i N_m - \sum_{m=1}^{i-1} N_m}$$

•Phase defect containment efficiency

$$PDCE_i = \frac{N_{ii}}{N_j}$$

•(Process) Defect removal efficiency

$$DRE = \frac{\sum_{i=1}^{k-1} N_i}{N}$$

Tecniche di V&V

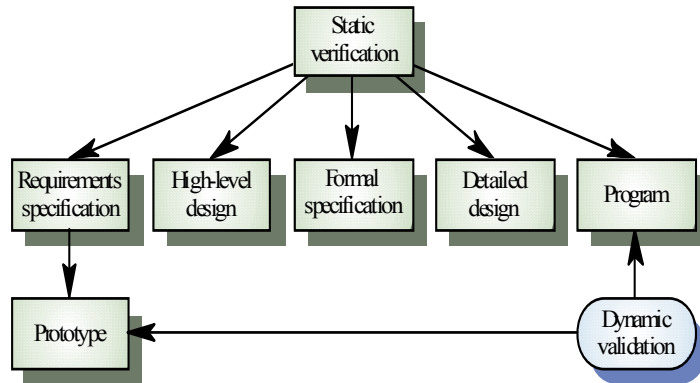
• Statiche

- inspections
- source code analysis

• Dinamiche (esecuzione di codice o modelli)

- testing

V&V statica e dinamica



17

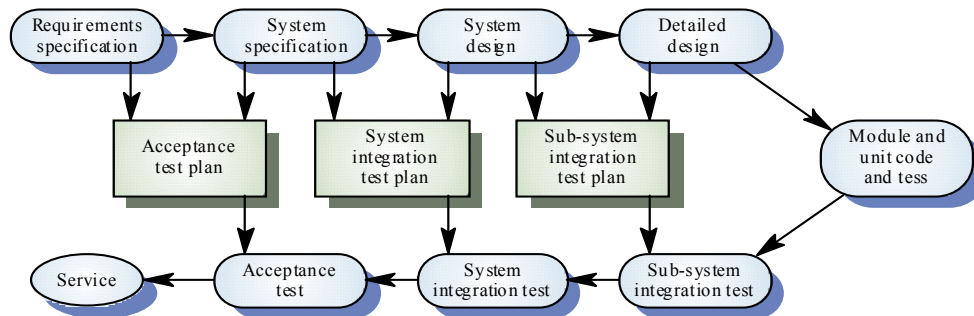
V & V pianificazione

- Occorre pianificazione attenta delle attività di V and V per ottenerne il massimo
- Deve essere fatta presto nel processo
- Deve definire il mix ideale di VV statica e dinamica
- La pianificazione del test si concentra più sui metodi da usare che sui test da eseguire

18

Il modello a V

• VV come thread parallelo a sviluppo



19

Il documento di software test plan

- Processo di test
- Requirements traceability
- Elementi da testare
- Calendario dei test
- Procedure di raccolta risultati
- Requisiti hardware e software
- Vincoli

20

Punti chiave

- **Verification e validation hanno obiettivi diversi**
 - Verification: controllo delle trasformazioni interne
 - Validation: controllo rispetto ai requisiti utente
- **V and V si basano su attività statiche e dinamiche (test)**
 - Il testing non è l'unica tecnica di V and V
 - Inspection e tecniche basate sulla lettura sono efficaci e necessarie
 - Il testing non può essere esaustivo
 - Il testing deve trovare errori (non dimostrare la correttezza)
- **V and V deve essere pianificata**

21

Storia

22

Preistoria

- **Tracce di testing nella preistoria della IS**
 - con la prima linea di codice è nato anche il testing ...
 - referenze bibliografiche risalgono al 1949
 - una definizione degli albori: ... il testing è ciò che un programmatore fa per individuare i bugs nei propri programmi...
 - è già nota la separazione concettuale fra DEBUGGING e TESTING ovvero fra attività di ricerca dei bugs ed attività di eliminazione degli stessi.
 - con il tempo le due attività saranno separate non solo nel tempo ma anche nello SPAZIO.

23

Anni 70

- **Negli anni 70 i primi seri tentativi di fornire dei fondamenti teorici ed approcci sistematici**
- **.. il sogno del testing esaustivo ... e le relative impraticabili definizioni ad esempio:**
 - l'obiettivo del testing è quello di dimostrare la correttezza dei programmi
- **... quindi un test perfetto si ha quando il programma non contiene errori**
- **Code Inspection di Fagan: efficaci, ma per anni sempre usate meno di quel che si dovrebbe**

24

Anni 70

- Goodhough and Gerhard "Toward a theory of test data selection" IEEE TSE SE-1(2) pp. 156-173, 1975
- Notevoli conquiste non solo sul piano teorico, ricca produzione di metodi e tecniche classificazioni e toponomastica, modelli per la rappresentazione dei programmi classificazione di errori ecc.

Anni 80

• Gli anni 80 eliminano i sogni, consolidano un patrimonio di riferimento; i risultati acquisiti aprono ad una visione: il testing come processo complesso dai costi estremamente elevati.

- il clima della fine anni 70 e' riflesso in G. Myers "The art of software testing" John Wiley and Sons N.Y. 1979
- esempio di consolidamento B. Beizer "Software testing techniques" Van Nostrand Reinhold N.Y. I ed. 1983 II ed. 1990
- due standard IEEE std 829-1983, std. 1008-1987; importante per le definizioni anche lo standard 610.12 1990 (già 729 1983)

Anni 80

Nuovo punto di vista

- Il testing e' il processo di esecuzione di un programma con l'obiettivo di trovare gli errori
 - Il programma fa quel che deve fare
 - Il programma **non** fa quel che **non** deve fare
- Un test che non rivela errori e' un test fallito

Test come processo indipendente dalla programmazione

- la progettazione, pianificazione ed implementazione del processo di testing inizia con i primi passi di un qualsiasi processo di produzione, manutenzione evoluzione del software

Anni 90

• Non solo test nella QA, VV

- consolidamento del rapporto fra testing e qualità
- più profonda conoscenza e miglioramento dei processi di testing
- nascono nuove sfide legate ai nuovi paradigmi di programmazione, produzione manutenzione ed evoluzione del software.

Fattori implicanti la qualità

Modelli costruttivi di qualità'

Non tutti i programmi devono avere tutte le qualità'

La determinazione delle proprietà che devono essere garantite ad un prodotto software dipende da fattori quali:

- Il tipo di applicazione;
- L'ambiente in cui il software viene utilizzato;
- Il modo di integrazione con l'ambiente;
- La vita prevista per l'applicazione;
- L'evoluzione prevista.

Anni 90

Il testing e' l'analisi e l'esecuzione sotto condizioni controllate dei vari componenti di un progetto software con l'obiettivo di mettere in evidenza difetti di qualità

•esemplare l'affermazione di Hetzel: " il testing e' pianificazione, progettazione, costruzione manutenzione e realizzazione di test ed ambienti di test"

Problemi aperti

- Costi di VV (anche 40-60% del costo totale di produzione)**
- VV di sistemi object oriented**
- VV in manutenzione ed evoluzione**
- come aumentare i livelli di automazione dei processi di testing**
- Test e certificazione di COTS**
- Test e certificazione di componenti Open Source**
- Test e certificazione di servizi e SLA**

VV vs ISO 9126

• Iso 9126 – come fare V&V di:

- Funzionalità'
 - Affidabilità'
 - Usabilità'
 - Efficienza
 - Manutenibilità'
 - Portabilità'
- Per lo piu' validazione
– Qualità' in uso escluso
– Qualità' interne anche

Funzionalità

- **Tendenzialmente test**
- **Verifica statica come attività preliminare**
- **Checklist (rispetto ai requisiti)**
 - Appropriatelyzza (tutte e sole le funzionalità)
 - Interoperabilità (soluzioni adottate)
 - Sicurezza (soluzioni adottate)
 - Aderenza alle prescrizioni
- **Test per accuratezza**

33

Affidabilità

- **Tendenzialmente test**
- **Verifica statica come attività preliminare**
- **Checklist (rispetto ai requisiti)**
 - Tolleranza ai guasti (guasti tollerati)
 - Recuperabilità (soluzioni adottate)
 - Aderenza alle prescrizioni

34

Usabilità

- **Test con utenti finali**
- **Verifica statica come attività complementare**
- **Checklist (rispetto ai manuali)**
 - Comprensibilità
 - Apprendibilità
 - Aderenza alle prescrizioni
- **Questionari all'utenza (a seguito di prove)**
 - Operabilità
 - Attraenza

35

Efficienza

- **Test**
- **Verifica statica come attività preliminare**
- **Checklist (risp. a criteri realizzativi)**
 - Efficienza algoritmica
 - Allocazione/deallocazione delle risorse
- **Miglioramento vs. confidenza**
 - L'efficienza deve essere provata
 - La verifica statica non dà confidenza, ma migliora il codice

36

Manutenibilità

- **Verifica statica come strumento ideale**
- **Checklist (coding standards)**
 - Analizzabilità
 - Modificabilità
 - Aderenza alle prescrizioni
- **Checklist (batterie di prove da eseguire)**
 - Verificabilità

37

Portabilità


- **Verifica statica**
- **Checklist (coding standards)**
 - Adattabilità
 - Aderenza alle prescrizioni
- **Prove come strumento complementare**
 - Installabilità
 - Coesistenza
 - Rimpiazzabilità

38

Testing

39

Tecniche di V&V

- **Statiche**
 - inspections
 - source code analysis
- **Dinamiche (esecuzione di codice o modelli)**
 - testing 

40

Testing

- **Verifiche (o validazioni) dinamiche**
 - Attività che prevedono l'esecuzione del software
 - In un ambiente controllato e con input e output definiti
 - Come verifica (sui moduli)
 - Come validazione (sul sistema)
- **Metodo tanto intuitivo quanto complesso**
 - Progettazione, esecuzione, analisi, debugging
 - Validazione dei risultati, terminazione delle prove
- **Costoso**
 - Occorrono molte risorse (tempo, uomini, macchine)
 - È necessario un processo definito
 - Richiede ulteriori attività di ricerca del difetto e correzione

41

Tipi di testing

- **Defect testing**
 - Obiettivo: trovare difetti
 - Un buon test case trova un difetto
- **Statistical testing**
 - Obiettivo: stima dell'affidabilità
 - Un buon test case (suite) riflette le condizioni operative (frequenza dell'input in condizioni di uso)

42

Defect testing

43

Test e fasi

- **Component/unit testing**
 - Testing di elementi del sistema
- **Integration testing**
 - Testing di gruppi di elementi integrati a creare sub-system (o system)
- **System testing**
 - Testing del sistema software nella sua completezza, considera proprietà globali
 - *test di stress* : proprietà del sistema in condizioni di sovraccarico,
 - *test di robustezza* : proprietà del sistema se i dati non sono corretti,
 - *test di sicurezza* : proprietà di sicurezza del sistema.
- **Acceptance testing**
 - Come system testing, ma fatto dal cliente o su dati del cliente o su piattaforma del cliente
- **Regression testing**
 - Esecuzione di test già eseguiti dopo modifiche o aggiunte

44

Linee guida

- **Solo il test esaustivo puo' dimostrare l'assenza di difetti di un programma**

- Ma il test esaustivo e' impossibile perfino su casi triviali

- **Vi sono vari approcci al test**

- Funzionalita' (vista esterna) vs. struttura e componenti (vista interna)
- Parti (funzioni o componenti) vecchi vs. nuovi
- Casi tipici vs. casi estremi

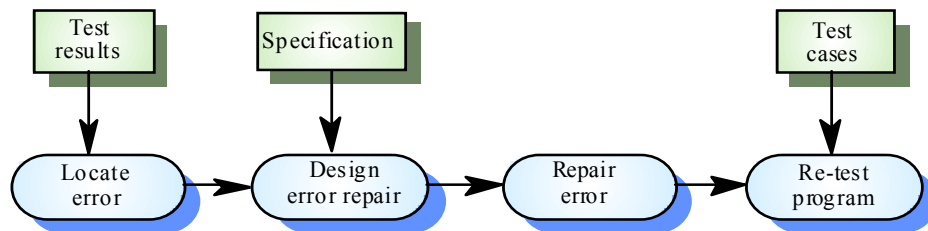
45

Testing e debugging

- **Defect testing e debugging sono processi distinti**
- **V and V (e quindi testing) ha per obbiettivo la ricerca di difetti**
- **Debugging cerca e rimuove la causa di un difetto**
 - Ipotesi sulla causa
 - Test dell'ipotesi

46

Processo di debugging



47

Defect testing Component (unit/module) Testing

48

Criteri di progetto test

•Black box (funzionale)

- Non considera come e' fatto internamente l'elemento da testare
 - Equivalence partitioning
 - Boundary conditions
- The program test cases are based on the system specification
- Test planning can begin early in the software process

•White box (strutturale)

- Considera come e' fatto internamente l'elemento da testare (struttura)
- Basato sulla nozione di copertura, i. e. sull'esecuzione di elementi del grafo di controllo di un programma
 - Copertura di
 - » Nodi
 - » Condizioni
 - » Decisioni (branches)

•Questi due criteri sono complementari tra loro e ognuno può rilevare malfunzionamenti non rivelabili con l'altro.

49

Terminologia

•Test data

- *Input per il sistema, scelto per testare il sistema*

•Test case

- *tripla <input, output, ambiente>*

•Test suite

- *Un insieme (una sequenza) di test case*

•Procedura di test

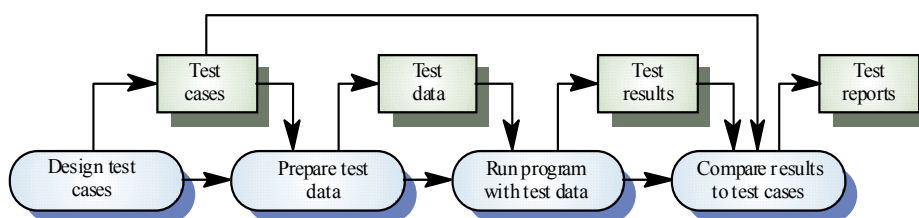
- *Procedura (automatica e non) per eseguire, registrare analizzare e valutare i risultati di una test suite*

•Oracle

- *Predice il risultato corretto (output) dato un input*

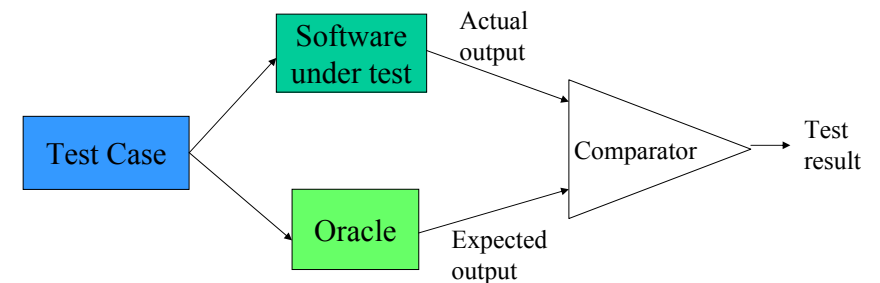
50

Processo di defect testing



51

Oracolo (Oracle)



52

Oracolo

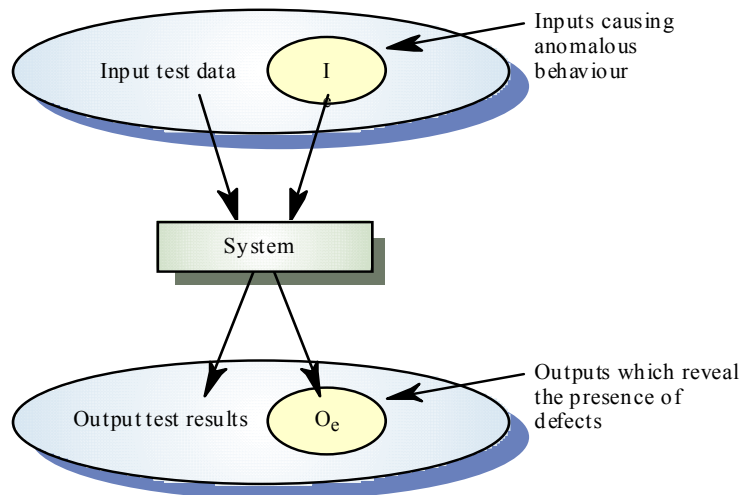
- **Oracolo**
 - Fornisce il risultato corretto per un test data
- **Oracolo umano**
 - Si basa su specifiche o judgement
 - Può sbagliare
- **Oracolo automatico**
 - Derivato da specifiche formali
 - che possono essere sbagliate (validation)
 - Derivato da versione precedente del programma
 - Derivato da sistema software sviluppato da altri

53

Defect testing Component (unit/module) Testing Black box

54

Black-box testing



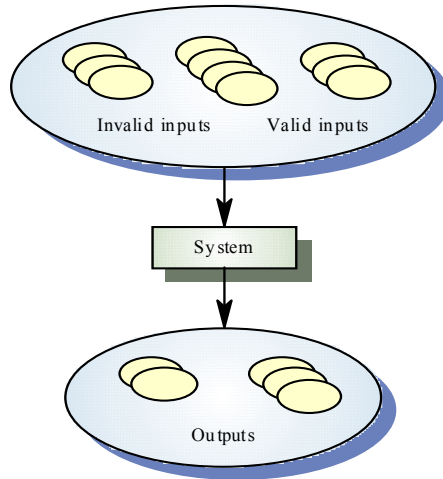
55

Black box (functional)

- **Testare almeno ogni funzione**
- **Per una singola funzione:**
 - Criteri di selezione test
 - equivalence classes
 - Boundary condition
 - Criteri di terminazione
 - 1 test per equivalence class/boundary condition
 - n>1 test per equivalence class/boundary condition
 - » Aumentare la confidenza

56

Equivalence classes



57

Equivalence classes

•Premesse

- Test esaustivo impossibile
- Scegliere test case che massimizzino la probabilita di trovare difetto

•Conseguenza

- Considerare valori di input per cui il comportamento del programma si possa supporre analogo (classe di equivalenza)
- Scrivere almeno un test case per classe

58

Equivalence classes

- **Subset dei valori possibili di input (con associati valori di output) per cui la funzione si comporta allo stesso modo**
- **Una classe corrisponde ad un set di input valido o invalido definito da una condizione sulle variabili di input**
 - Valido: quel che la funzione deve fare
 - Non valido: quel che la funzione **non** deve fare
- **Scegliere almeno un test case per classe (valida – non valida)**

59

Equivalence classes

•Passi

- 1 identificare le classi, valide e non valide
- 2 definire test case(s) per ogni classe

60

Input e classi

- **Input = Intervallo di valori**
 - 'aliquota IVA tra 0 e 20'
 - Classe di valori validi entro intervallo (0 a 20)
 - Classe di valori non validi < intervallo (<0)
 - Classe di valori non validi > intervallo (>20)
- **Input = numero di elementi associati**
 - 'auto puo' avere da 1 a 6 proprietari'
 - Classe per numero valido (1 a 6 proprietari)
 - Classe di numero < numero valido (0 proprietari)
 - Classe di numero > numero valido (>6 proprietari)

61

Input e classi

- **Input = set di valori**
 - 'veicolo puo' essere Auto o Moto'
 - Classe per ogni valore (se trattati in modo diverso): Auto, Moto
 - Classe per valore non valido: Bici
- **Condizione 'must be' su input**
 - 'Primo carattere di identificatore deve essere lettera'
 - Classe per must be rispettato: lettera
 - Classe per must be non rispettato: non lettera
- **Spezzare classe in classi piu' piccole se la funzione potrebbe trattarla diversamente**
 - 'ricerca elemento in array'
 - Classe per elemento presente
 - Classe elemento presente, ripetuto in array
 - Classe elemento presente, non ripetuto
 - Classe per elemento non presente

62

Boundary condition

- **Test case sul limite delle classi sono spesso efficaci**
- **Boundary conditions**
 - Il limite della classe
 - Immediatamente sopra o sotto il limite rispetto a
 - Classi di input
 - Classi di output
- **Possano essere condizioni molto sottili**
 - Richiedono creativita'

63

Esempio

- $\text{abs}(x) := (x \geq 0 \text{ then } y = x \text{ else } y = -x)$
 - equivalence classes: x positivo (valida), x negativo (valida), x non numerico (non valida)
 - boundary condition: x (vicino a) zero
 - Terminazione: due per classe
 - Test cases
 - » T1, 1, 1 T2 10,10
 - » T3, -1, 1 T4 -10, 10
 - » T5, 0.001, 000.1 T6, -0.001, 000.1
 - » T7, 'x', error

64

Esempio

- $\text{sqrt}(x) := (x \geq 0 \text{ then } y = \sqrt{x})$
 - equivalence classes: x positivo (valida), x negativo (non valida), x non numerico (non valida)
 - boundary condition: x (vicino a) zero
 - Terminazione: una per classe
 - Test cases
 - » T1, 1, 1 T2 4,2
 - » T3, -1, error T4 -2, error
 - » T5, 0.0001, 0.01 T6, -0.001, error
 - » T7, 'x', error

65

Search routine - specifica

```
procedure Search (Key : ELEM ; T: ELEM_ARRAY;  
Found : in out BOOLEAN; L: in out ELEM_INDEX) ;
```

Pre-condition

-- the array has at least one element
T'FIRST <= T'LAST

Post-condition

-- the element is found and is referenced by L
(Found and T (L) = Key)

or

-- the element is not in the array
(**not** Found **and**
not (exists i, T'FIRST >= i <= T'LAST, T (i) = Key))

66

Search routine - classi

- **Rispetto alle pre/post condizioni**
 - Pre
 - Array has at least one element
 - Not (Array has at least one element)
 - Post
 - Input presente nell'array
 - Input non presente nell'array
- **Pre condizione non specificata**
 - Array is ordered or not
 - Array has 2 or more elements

67

Search routines - classi

- **Rispetto agli elementi dell'array (pre)**
 - Zero elementi
 - Un elemento
 - Piu' di un elemento
 - Se piu' di uno, testare con varie cardinalita'
 - Accedere elemento primo, medio e ultimo

68

Search routine

Array	Element
Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

Input sequence (T)	Key (Key)	Output (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

69

Defect testing Component (unit/module) Testing White box

70

White box

- **Criteria di selezione**
 - statement coverage (copertura)
 - branch coverage
 - condition coverage
 - path coverage
- **Criteria di terminazione**
 - 1 test per coverage
 - n>1 test per coverage

71

Misure di efficacia

- **La efficacia di un test case o test suite e' data da Test Effectiveness Ratio**

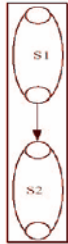
$$\text{TER} = \frac{\# \text{ elements activated}}{\# \text{ elements in test item}}$$

72

Costruzione del grafo di controllo

A un programma imperativo sequenziale si associa un grafo con queste regole

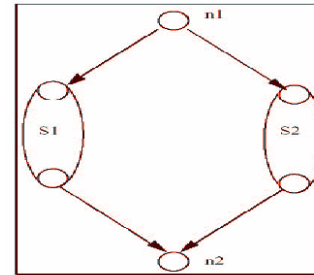
- Ad un comando di *assegnamento* o di ingresso/ uscita si assegna un nodo
- Una sequenza di comandi e' rappresentata da un arco
 - S1; S2



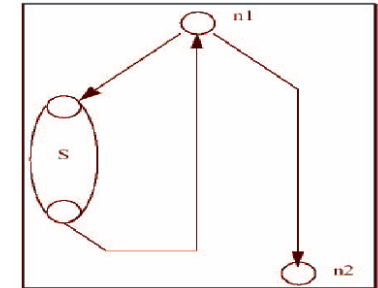
73

- Ad un comando di controllo si assegnano nodi e archi come segue

- If (condition) S1 else S2;



- while (condition) do S;

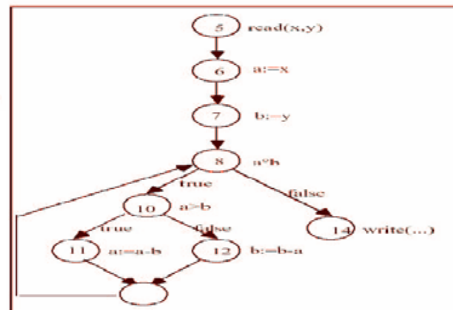


74

Esempio

```

1  program gcd (input, output);
2  var
3  x, y, a, b: integer;
4  begin
5  read (x, y);
6  a := x;
7  b := y;
8  while a <> b do
9  begin
10     if a > b
11     then a := a - b;
12     else b := b - a
13     end;
14  write ('MDC dei dati e''', a)
15  end.
    
```



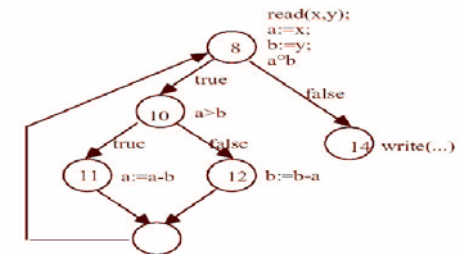
75

Variazioni del grafo

• Il grafo corrispondente ad un programma può essere modificato a seconda dell'uso che se ne intende fare

• Possono essere sia eliminati che aggiunti nodi ed archi

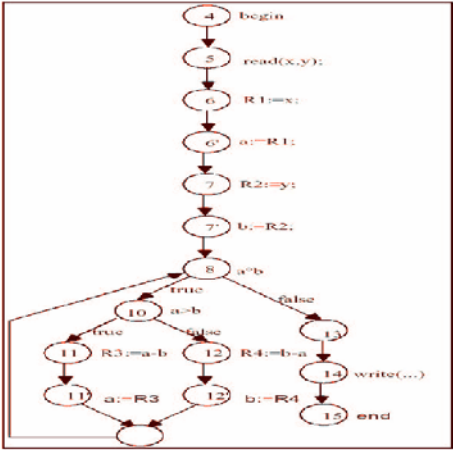
- *Esempio* : Se si e' interessati solo ai comandi che comportano più alternative nel flusso di controllo i nodi corrispondenti a sequenze di ingressi/ uscite ed assegnamenti possono essere collassati:



76

Statement coverage

• *Esempio* : Rappresentazione delle diverse azioni che corrispondono all'assegnamento (R1, ..., R4, registri):

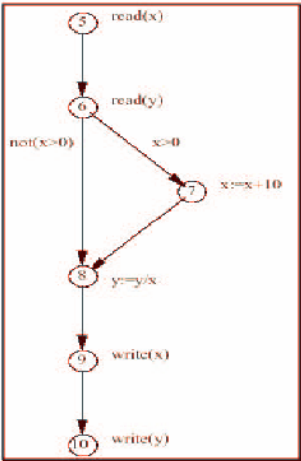


- Un test T soddisfa il criterio di copertura dei comandi (statement) se e solo se ogni comando eseguibile del programma è eseguito per almeno un dato di test in T.
- Nel grafo di controllo del programma, *ogni nodo* corrispondente ad un comando eseguibile deve essere percorso *almeno una volta*
- Misura di copertura C :

$$C = \frac{\text{numero di comandi eseguiti}}{\text{numero totale di comandi eseguibili}}$$

```

1 Program statement (input, output);
2 var
3   x,y : real;
4 begin
5   read(x);
6   read(y);
7   if x > 0 then x:=x+10;
8   y:=y/x;
9   write(x);
10  write(y);
11 end.
    
```



- Il test S = {(x = 20, y = 30)}, causa l'esecuzione di tutti i comandi del programma e quindi soddisfa il criterio di copertura dei comandi (la copertura assicurata da S è 1)
- Il malfunzionamento (divisione per zero), che si verifica quando il comando alla linea 8 viene eseguito con valore zero per la variabile x non è rilevato dal test S
- Il test S non comporta l'esecuzione del programma per tutti i valori delle decisioni: la decisione alla linea 7 (x > 0) dà valore *vero* per tutti i dati di test contenuti in S

Branch coverage

•Un test T soddisfa il criterio di copertura delle decisioni (branch) se e solo se ogni arco nel grafo di controllo del programma e' eseguito almeno una volta

•Misura di copertura C :

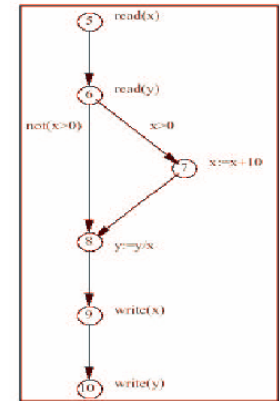
numero di archi eseguiti /
numero totale di archi eseguibili

Ovviamente, se un test soddisfa il branch test allora soddisfa anche il statement test, ma non è detto il viceversa.

81

•Il test $S = \{(x = 20, y = 30)\}$ non soddisfa il criterio di copertura delle decisioni: l'arco (6, 8) non viene mai percorso

• Il test $D = \{(x = 20, y = 30), (x = 0, y = 30)\}$ soddisfa il criterio di copertura delle decisioni, in quanto causa la percorrenza di tutti gli archi almeno una volta, e permette di rilevare il malfunzionamento causato dalla divisione per zero alla linea 8



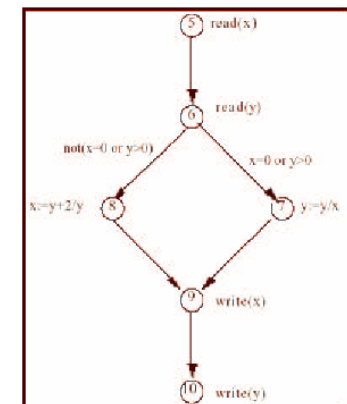
82

```
1 Program branch (input, output);
2 var
3   x,y:real;
4 begin
5   read(x);
6   read(y);
7   if (x = 0 or y > 0) then y:=y/x
8   else x:=y+2/y;
9   write(x);
10  write(y);
11end.
```

83

•Il test $\{(x = 5, y = 5), (x = 5, y = -5)\}$ soddisfa il criterio di copertura delle decisioni, ma non rileva un malfunzionamento causato da una divisione per zero sia alla linea 7 che alla linea 8

• La decisione alla linea 7 può essere posta a *vero* ed a *falso* agendo unicamente su una sola delle due condizioni in *or* nella decisione



84

Condition coverage

• Un test T soddisfa il criterio di copertura delle condizioni se e solo se *ogni singola sotto-condizione* che compare nelle decisioni del programma *assume il valore vero e falso* per un qualche dato di test in T .

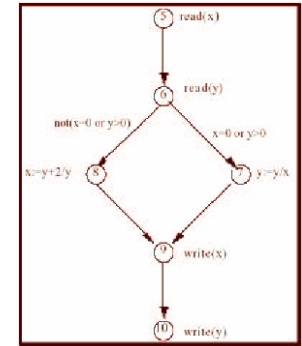
85

• Il test $C = \{(x = 0, y = -5), (x = 5, y = 5)\}$ soddisfa il criterio di copertura delle condizioni, infatti:

- per il dato $(x = 0, y = -5)$ la condizione $(x = 0)$ vale *vero* e la condizione $(y > 0)$ vale *falso*,
- per il dato $(x = 5, y = 5)$ la condizione $(x = 0)$ vale *falso* e la condizione $(y > 0)$ vale *vero*

• Il test C rileva il malfunzionamento causato dall'anomalia alla linea 7, ma non quello causato dall'anomalia alla linea 8

• Il test C non soddisfa il criterio di copertura delle decisioni, infatti per entrambi i dati del test C la decisione di linea 7 vale *vero*)



86

Condition + branch coverage

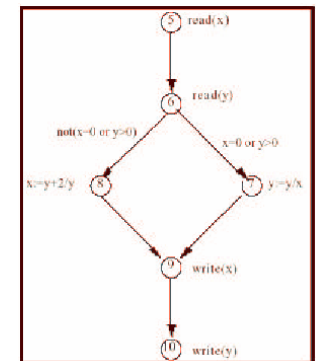
• Un test T soddisfa il criterio di copertura delle condizioni e delle decisioni se e solo se ogni decisione vale sia vero che falso ed ogni singola sotto-condizione che compare nelle decisioni del programma vale sia vero che falso per diversi dati di test in T

• Il criterio di copertura delle decisioni e delle condizioni contiene sia il criterio di copertura delle condizioni che il criterio di copertura delle decisioni

87

• Il test $C = \{(x = 5, y = -5), (x = 0, y = -5), (x = 5, y = 5)\}$ soddisfa il criterio di copertura delle condizioni e delle decisioni

• Il test C rileva il malfunzionamento causato dall'anomalia alla linea 7, e anche quello causato dall'anomalia alla linea 8



88

Path coverage

- Un test T soddisfa il criterio di copertura dei cammini se e solo se ogni cammino nel programma viene percorso per almeno un dato di test in T
- Misura di copertura C :
numero di cammini percorsi /
numero totale di cammini percorribili
- Il numero di cammini eseguibili può essere infinito, rendendo tale criterio non applicabile
- Per limitare il numero di cammini si fissa il numero massimo di percorrenze di ciascun ciclo

89

n-coverage cycles

- Un test T soddisfa il criterio di n- copertura dei cicli se e solo se ogni cammino contenente un numero di iterazioni di ogni ciclo non superiore ad n è eseguito per almeno un dato di test in T

90

Teorema Weyuker

- Dato un generico programma P , l'esistenza di un dato di test tale da causare l'esecuzione
 - di una particolare istruzione di P , oppure
 - di una particolare condizione di P , oppure
 - di un particolare cammino di P , oppure
 - di ogni istruzione di P , oppure
 - di ogni condizione di P , oppure
 - di ogni cammino di P
- non è decidibile.**

91

Indecidibilita'

- Il settore del testing e' tormentato da problemi indecidibili
- un problema e' detto indecidibile (irrisolvibile) se e' possibile dimostrare che non esistono algoritmi che lo risolvono ... (macchina di turing, halting problem, teorema della esistenza di funzioni non calcolabili, etc...)
 - Es. stabilire se l'esecuzione di un programma termina a fronte di un input arbitrario e' un problema indecidibile
 - Es. stabilire se due funzioni sono equivalenti
 - Es. Definire il criterio di test ideale di un programma

92

Esercizio

```
double abs(double x){
  if (x>=0) then return x; else return -x;
}
```

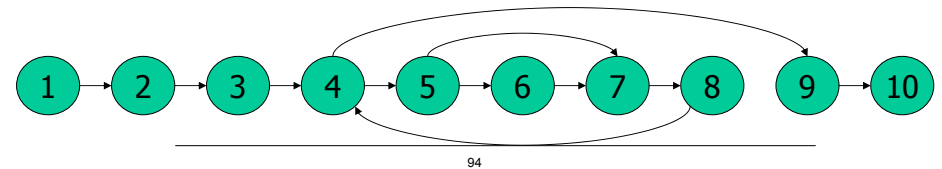
- Definire test per
 - statement coverage
 - branch coverage
 - path coverage

93

Esercizio

```
1 float homeworkAverage(float[] scores) {
2   float min = 99999;
3   float total = 0;
4   for (int i = 0; i < scores.length; i++){
5     if (scores[i] < min)
6       min = scores[i];
7     total += scores[i];
8   }
9   total = total - min;
10  return total / (scores.length - 1);
11 }
```

- Definire test per
 - statement coverage
 - branch coverage
 - condition coverage
 - path coverage



94

Defect testing Integration Testing

95

Integration testing

- **Applicato all'integrazione di due o piu' componenti/ moduli**
 - Definizione di modulo dipende dal contesto, dai casi
- **Obiettivo: trovare difetti che nascono dall'interazione dei moduli, su funzioni cross-module**
 - Assumendo che i moduli siano singolarmente 'senza difetti'
 - Aims at finding faults concerning the interoperation of units and the aggregate whole functions
- **Puo' esser fatto da team di sviluppo o team separato**
- **Strategie:**
 - Top-down vs. Bottom-up
 - Big-bang vs. Incrementale

96

Terminologia

- **Driver** componente fittizia per pilotare un componente
- **Stub** componente fittizia per simulare un componente

- **Mock**

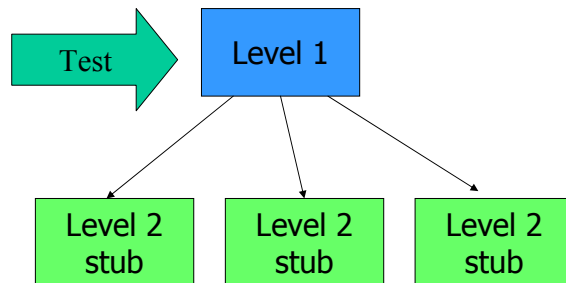
97

Big-bang vs. Incrementale

- **Big bang:** tutti I moduli sono integrati in un colpo solo
- **Incrementale:** moduli integrati uno ad uno. Vantaggi
 - Più facile identificare causa di difetto (ultimo modulo aggiunto)
 - Problemi con interfacce identificati prima
 - I primi moduli integrati sono testati più a lungo

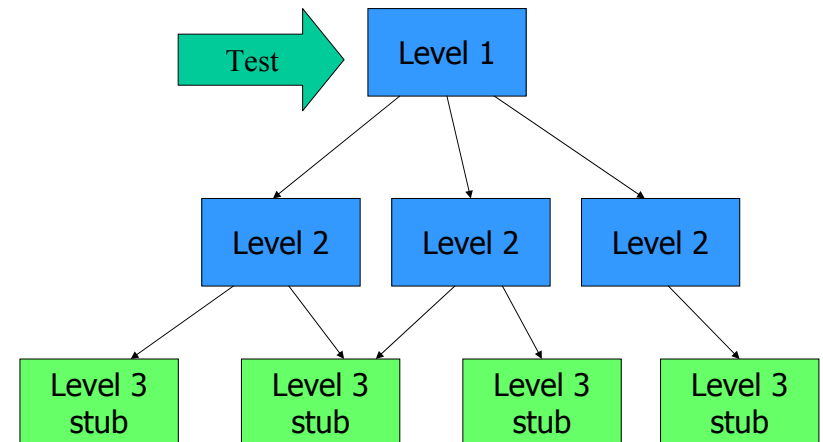
98

Top-down testing



99

Top-down testing



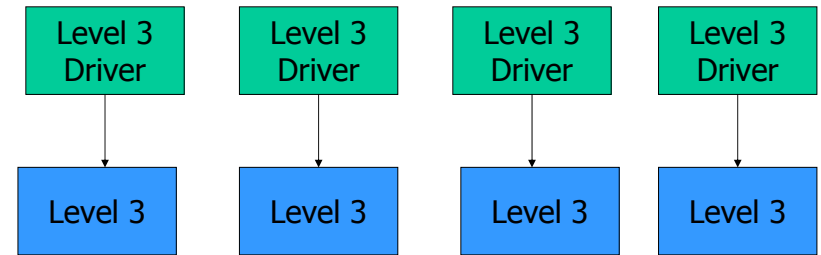
100

Top-down testing

- **Pro**
 - Permette di scoprire presto problemi architetturali
 - Fornisce un sistema 'funzionante' molto presto
- **Contro**
 - Obbliga a definire stub per tutti i livelli bassi
 - Implica sviluppo top down

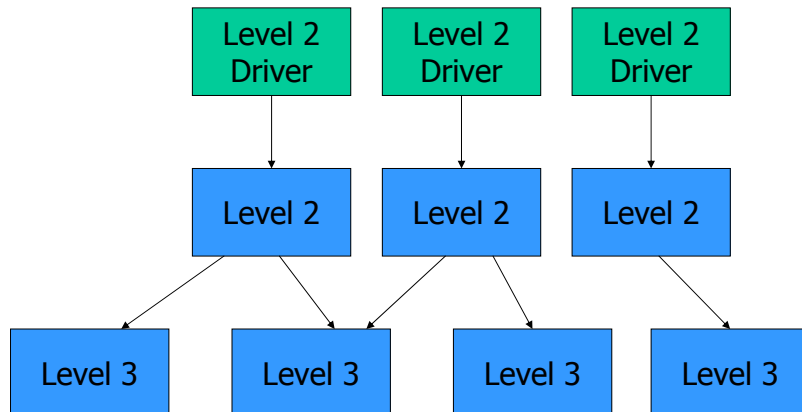
101

Bottom-up testing



102

Bottom-up testing



103

Bottom-up testing

- **Pro**
 - Scrittura dei driver di basso livello piu facile
 - Rispetto scrittura di stub dei livelli bassi
- **Contro**
 - Implica sviluppo bottom up
 - Obbliga a definire drivers per tutti i livelli bassi
- **In pratica si usa un mix di top down e bottom up**

104

Esempio

•Azienda ha vari impiegati e dipartimenti, ogni impiegato appartiene a un dipartimento

•Gestione stipendi

- Stipendio(ID) – rende stipendio dato ID di impiegato
- Bonus(cifra) – da bonus 'cifra' per anno in corso a impiegati di dipartimento con venditeAnnuali maggiore

•Impiegato

- Nome, cognome, ID, stipendiobase

•Dipartimento

- Nome, id, venditeAnnuali

•ElencoImpiegati

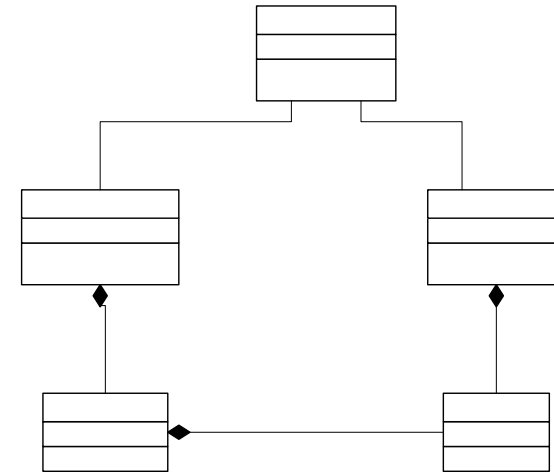
- Aggiunge impiegato
- Rende impiegato dato ID

•ElencoDipartimenti

- Aggiunge dipartimento
- Rende dipartimento dato ID

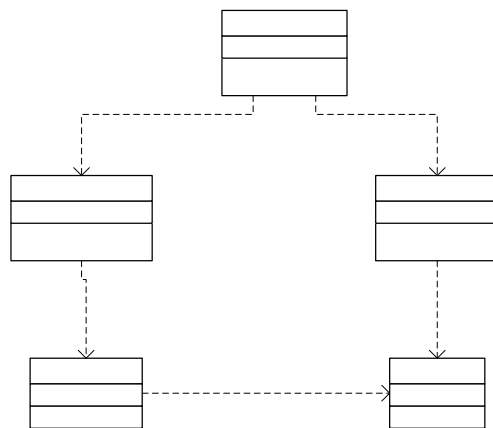
105

Class diagram (1)



106

Dependencies (1)



107

Integration

•Bottom up

- Impiegato
- ElencoImpiegati Dipartimento
- ElencoDipartimenti
- GestioneStipendi

•Top Down

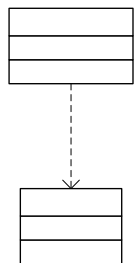
- GestioneStipendi
- ElencoDipartimenti ElencoImpiegati

+stipendio()

108

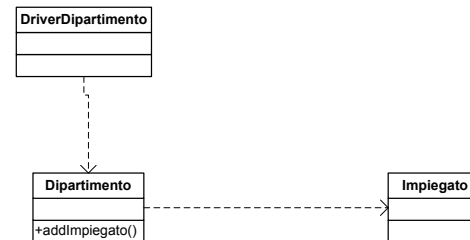
1

BU - 1



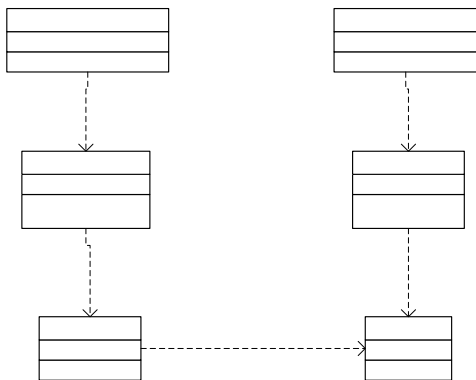
109

BU - 2



110

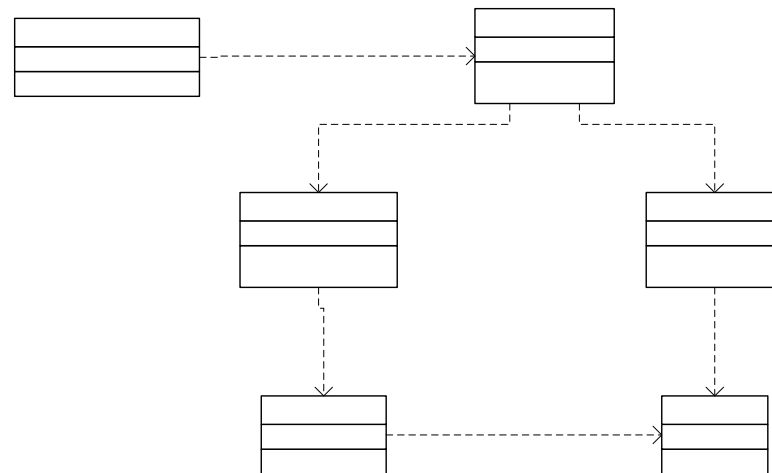
BU - 3



111

DriverElencoI

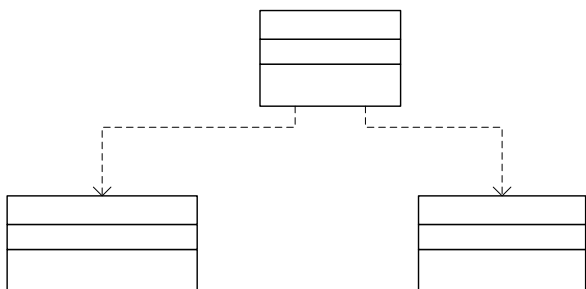
BU - 4



DriverGestioneStipendi

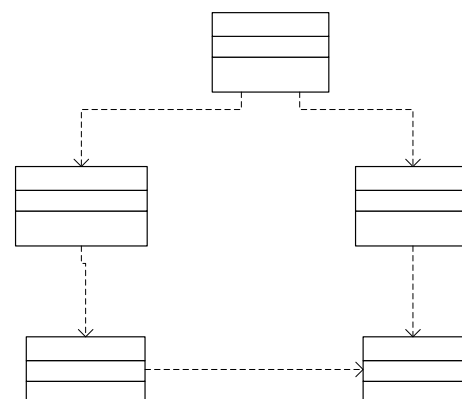
Impiegato

TD - 1



113

TD - 2

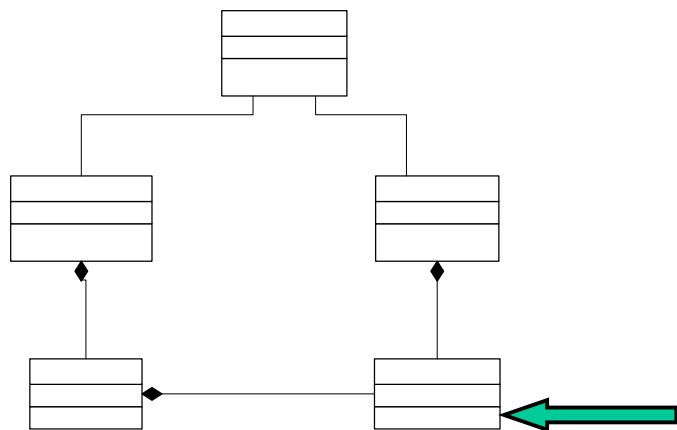


- StubDipartimento e StubImpiegato probabilmente inutili, poiche' di complessita' analoga a Dipartimento e Impiegato

114

GestioneStipendi

Class Diagram (2)

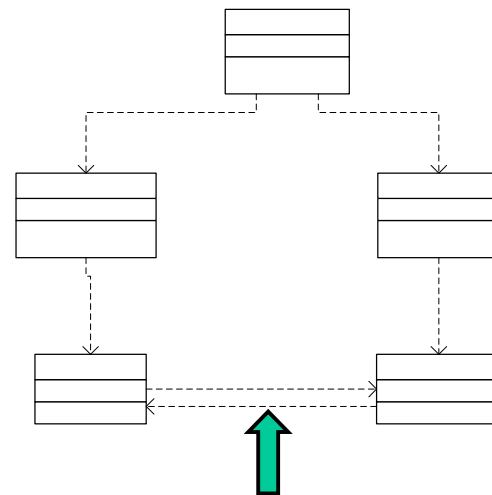


115

+cercaDipartimento()

er

Dependencies (2)



116

+cercaImpiegato()

ar

no

tin

In

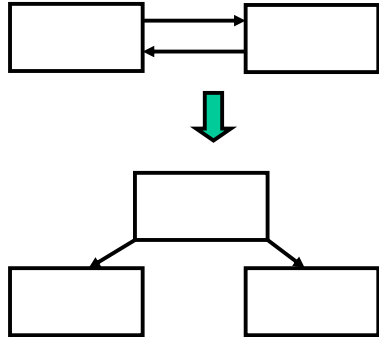
te

Integration

•1 Considerare le classi con dipendenza reciproca come una sola

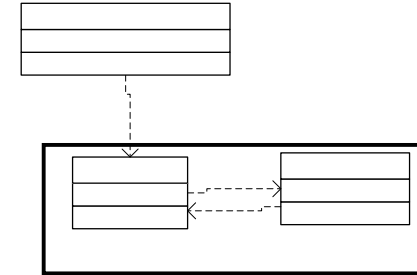
- Problema se classi grandi / alte nella gerarchia di dipendenza

•2 Modificare il design, spezzare il ciclo



117

Caso 1 - BU



- Passi seguenti come per BU senza dipendenza ciclica tra Dipartimento e Impiegato

118

DriverIn

Test – Visione d’insieme

119

Cosa, come e quando

- **Test di modulo (componente, package)**
 - Copertura di ogni metodo pubblico del package (bb)
- **Test di modulo (classe)**
 - Copertura di ogni metodo (bb e/o wb)
 - Copertura di ogni requisito di modulo (bb)
- **Test di integrazione**
- **Test di accettazione**
 - Copertura (= almeno un test case per) di ogni requisito (bb)
 - Copertura di ogni scenario (bb)

120

Teoria del test

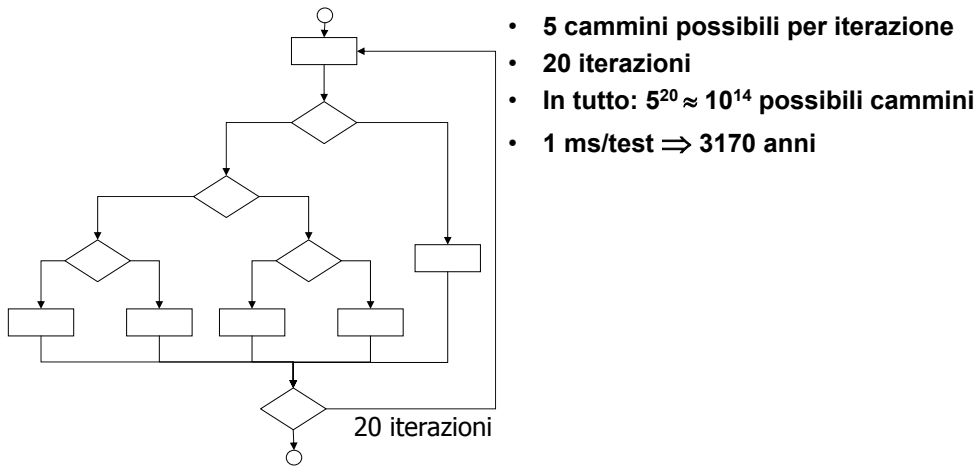
121

Test esaustivo

- funzione: $Y = A + B$
- A e B interi a 32 bit
- numero di test cases totale: $2^{32} * 2^{32} = 2^{64} \approx 10^{20}$
- 1 ms/test \Rightarrow 3000 miliardi anni

122

Test esaustivo



123

Tesi di Dijkstra

- Osservando i malfunzionamenti possiamo dedurre la presenza di difetti
- Ma: il testing non può dimostrare l'assenza di difetti, ma può solo dimostrare la presenza di difetti (Tesi di Dijkstra)
- Non vi è garanzia che se alla n-esima prova un modulo od un sistema abbia risposto correttamente (ovvero non sono stati più riscontrati difetti), altrettanto possa fare alla (n+1)-esima
- Impossibilità di produrre tutte le possibili configurazioni di valori di input (test case) in corrispondenza di tutti i possibili stati interni di un sistema software

124

Legge di Weinberg

- **Chi sviluppa un programma non e' adeguato a testarlo poiche'**
 - Apprezza il suo lavoro e tende a vederlo da un lato solo positivo
 - Se non ha considerato un problema continuera' a non considerarlo
- **Il Testing dovrebbe essere fatto da**
 - Team separato di QA
 - Peers

125

Legge Pareto-Zipf

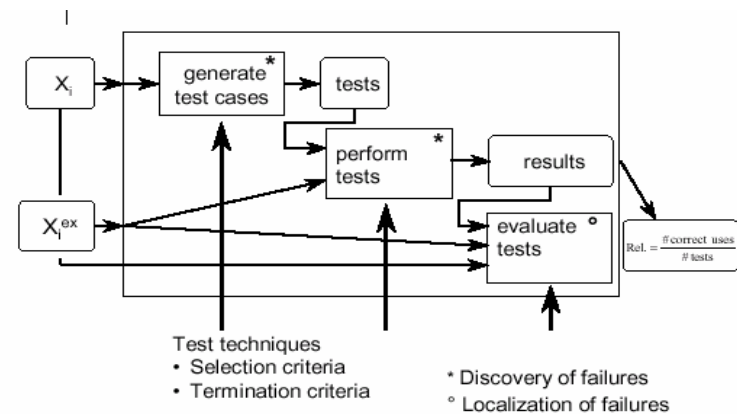
- **L' 80% dei difetti sono originati dal 20% dei componenti**
- **Concentrare il test (QA in generale) sui componenti piu' difettosi**

126

Test planning

127

Test process



128

Il processo di Test

- **Il testing è un processo;**
- **L'esigenza di definire modelli di riferimento a partire dai quali istanziare tali processi;**
- **Un modo per fissare riferimenti comuni per tali processi: definire i deliverable che debbono essere prodotti nelle fasi e con le attività che lo caratterizzano;**
- **I deliverable del processo: documenti (in forma cartacea o magnetica);**
- **IEEE Standard for Software Test Documentation (Std. 829-1998 – Revised Std. 829 1983).**

Documenti

- **Planning and specification documents**
 - **TP** Test Plan
 - **TDS** Test Design Specification
 - **TCS** Test Case
 - **TPS** Test Procedure Specification
- **Enactment documents**
 - **TTR** Test -item Transmittal Report
 - **TL** Test Log
 - **TIR** Test Incident Report
 - **TSR** Test -Summary Report

Documenti di planning

- **Test Plan (TP):** un documento che descrive l'oggetto, l'approccio generale, le risorse e lo schedule delle attività di testing da realizzare; fra l'altro identifica i Test Item, le caratteristiche da testare, le attività di testing, rischi e piani di emergenza, criteri generali di Pass/Fail.
- **Test Design Specification (TDS):** un documento che specifica per una o più (o per combinazioni di) caratteristiche da testare i dettagli dell'approccio al testing (tecniche di testing, analisi dei risultati, lista dei test case con loro motivazione ed attributi generali).
- **Test Item:** codice sorgente, codice oggetto, job control, data control del software da sottoporre a testing;
 - Un test item è accompagnato dalla relativa documentazione tecnica (requisiti, specifiche, progetto, etc.);
- **Pass/fail criteria:** regole di decisione da usare per stabilire se una caratteristica software supera o meno il test.

Documenti di planning

- **Test Case Specification (TCS):** è un documento che specifica un test case individuato da un TDS; il documento deve specificare gli input, i risultati attesi (oracolo), le condizioni di esecuzione (incluso hardware e software necessario);
- **Test Procedure Specification (TPS):** è un documento che specifica per uno o più test case i passi da fare per eseguirli; in particolare il documento deve descrivere come preparare l'esecuzione della procedura, come innescare e condurre tale esecuzione, quali rilevazioni e misure vanno fatte, come sospendere l'esecuzione del test in presenza di eventi imprevisti, come riprendere una esecuzione sospesa.
- **Test Case:** inputs, execution conditions, expected results;
- **Test Procedure:** detailed instructions for the set-up, execution, evaluation of results

Test plan

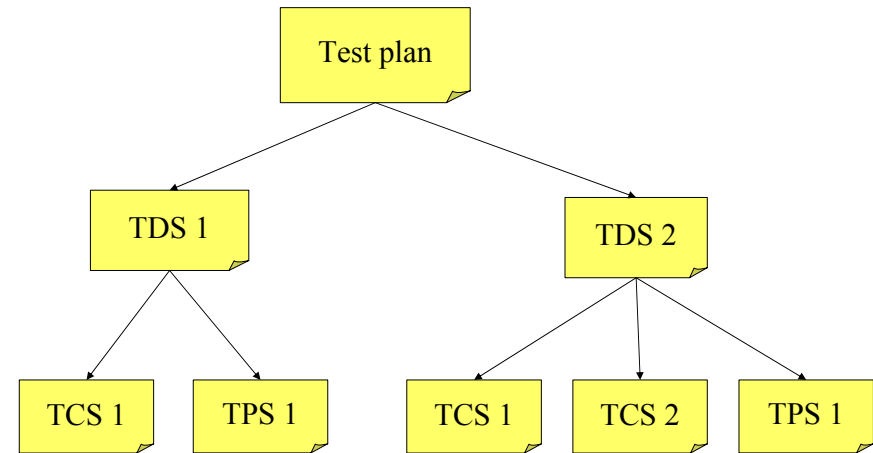
- Mappaggio test cases - requisiti

Test-requirement Correspondence Table

	Req x.1	Req x.2	Req y.1
Test 1	X	X	
Test 2	X		
Test 3			X

133

Relazioni tra documenti



134

Documenti di esecuzione

- **Test-Item Transmittal Report (TTR):** è un documento che deve accompagnare ogni software item consegnato al testing; è almeno costituito dalle informazioni di identificazione del software item, del suo stato, e della sua fisica allocazione;
- **Test Log (TL):** è la banca dati della memorizzazione sistematica, strutturata ed in ordine cronologico di tutti i dettagli rilevanti sulla esecuzione dei test; informazioni fondamentali di tale documento sono il successo o l'insuccesso dei test, l'occorrenza e la descrizione di eventi anomali e di test -incident;
- **Test Incident:** ogni evento occorso in un processo di testing e che richiede altre e più approfondite analisi ed investigazioni.

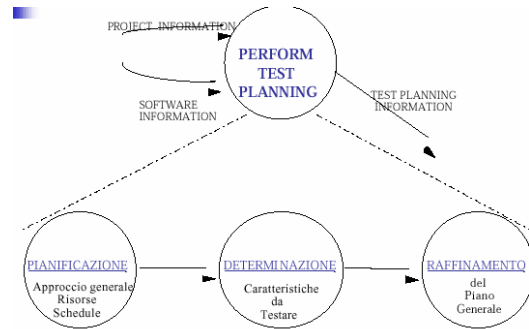
135

Documenti di esecuzione

- **Test-Incident Report (TIR):** è un documento che descrive i test -incident che si sono verificati; sono inclusi gli input, i risultati attesi, i risultati attuali, le anomalie, data e tempo, I tentativi di rieseguire il test, gli addetti al testing;
- **Test-Summary Report (TSR):** è un sommario ed una valutazione di una o più attività di testing; componenti fondamentali di tale documento sono la lista degli incidenti risolti e delle relative soluzioni, la lista degli incidenti irrisolti, una valutazione dei limiti del test
 - **Comprehensiveness:** una valutazione di quanto il test sia esaustivo rispetto agli obiettivi previsti nel piano (...vi sono caratteristiche non sufficientemente testate e le ragioni di ciò...);

136

Perform test planning



141

Pianificazione

INPUTS

- Piano progetto generale di Test
- Documentazione sui Requisiti

TASKS

- Specificazione dell'approccio generale al testing di unità
- Specificazione dei requisiti di completezza del testing
- Specificazione dei requisiti di normale terminazione
- Determinazione delle risorse richieste
- Specificazione dello schedule generale

OUTPUTS

- Informazioni sulla pianificazione di massima
- Richiesta delle risorse di massima

142

Caratteristiche da testare

INPUTS

- Documentazione sui requisiti dell'unità
- Documentazione sul progetto architettonale

TASKS

- Analisi dei Requisiti Funzionali
- Analisi degli altri Requisiti e delle procedure associate al software item
- Analisi degli stati (se specificati nei requisiti)
- Identificazione dei dati di I/O della unit
- Selezione degli elementi da includere nel testing (caratteristiche, procedure, stati e transizioni, caratteristiche dei dati, etc.)

OUTPUTS

- Lista elementi da includere nel test
- Richieste di chiarimento sui Requisiti

143

Raffinamento piano generale

INPUTS

- Lista elementi da includere nel Testing
- Informazioni sulla pianificazione di massima

TASKS

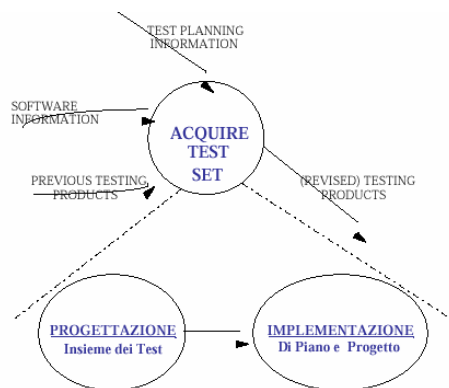
- Raffinamento degli approcci
- Specificazione richiesta risorse speciali
- Specificazione dello Schedule di dettaglio

OUTPUTS

- Piano dettagliato e finale di unit testing
- Documento richiesta di risorse speciali

144

Acquire test set



145

Progettazione test

INPUTS

- Lista elementi da includere nel Testing
- Piano di testing
- Documentazione Requisiti di Unità
- Documentazione Progetto Unità
- Specificazione Test precedenti (se eseguiti)

TASKS

- Progetto Generale (top-down e gerarchico) dell'insieme dei Test, ovvero architettura generale di Test Design Specification
- Test Procedures Specification (..approccio con riuso)
- Test Case Specification (..riuso..)
- Test Design Specification dettagliato e finale

OUTPUTS

- TDS
- TCS
- TPS

146

Implementazione

INPUTS

- Piano di testing
- TDS, TCS, TPS
- Documentazione Test Items
- Descrizione Data Structure
- Documentazione risorse di supporto
- Test data e test tools da riusare

TASKS

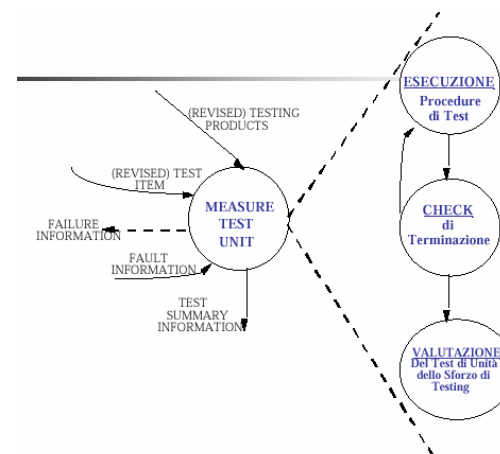
- Produzione, acquisizione e verifica test data
- Acquisizione risorse di supporto
- Allestimento finale Test Item

OUTPUTS

- Test Data
- Configurazione test item
- Risorse di supporto
- TCS, TPS
- Prime Informazioni di Sommario

147

Measure test unit



148

Esecuzione

INPUTS

- Test Data
- Test Item
- TDS, TCS, TPS
- Risorse di supporto
- Test data e test tools da riusare

TASKS

- Allestimento ambiente di Test ed esecuzione del test
- Determinazione dei risultati (inclusiva di sommario dei risultati, anomalie, incidenti, sospensione, eventuali modifiche a TDS, TCS, TPS, etc.) ed aggiornamento summary report

OUTPUTS

- LOG e summary report aggiornato
- TDS, TCS, TPS eventualmente modificati

149

Check terminazione

INPUTS

- Requisiti di completezza e terminazione
- LOG

TASKS

- Check di terminazione normale
- Check di terminazione anormale
- Eventuale Insieme di Test Supplementare
- Aggiornamento summary report e LOG

OUTPUTS

- LOG e summary report aggiornato
- TDS, TCS, TPS eventualmente modificati o aggiunti

150

Valutazione

INPUTS

- LOG (informazioni di esecuzione e check)
- TDS, TCS

TASKS

- Descrizione stato testing (in rapporto al piano e specificando eventuali variazioni)
- Descrizione Stato unità testata
- Produzione Summary Report Finale
- Collocazione in repository prodotti del testing

OUTPUTS

- Summary Report
- Repository Aggiornata

151

Statistical testing

152

Validazione dell'affidabilità (reliability)

- **Richiede di eseguire il programma per verificare il livello di affidabilità raggiunto**
- **Il tipo di test case da usare è diverso rispetto al defect testing**
 - Defect testing: test case per esercitare tutte le parti del programma
 - Obiettivo: trovare difetti
 - Statistical testing: test case devono rispecchiare l'uso del programma
 - Obiettivo: stima dell'affidabilità
 - Deve essere disponibile un set di test case che simulino (o provengano da) uso effettivo del programma

Statistical testing

- **Misurare il numero di difetti permette di predire il livello di affidabilità**
 - Per motivi statistici può essere necessario ottenere più difetti di quelli ammessi dalla specifica di affidabilità
- **Il programma viene testato e corretto finché si raggiunge il livello voluto di affidabilità**

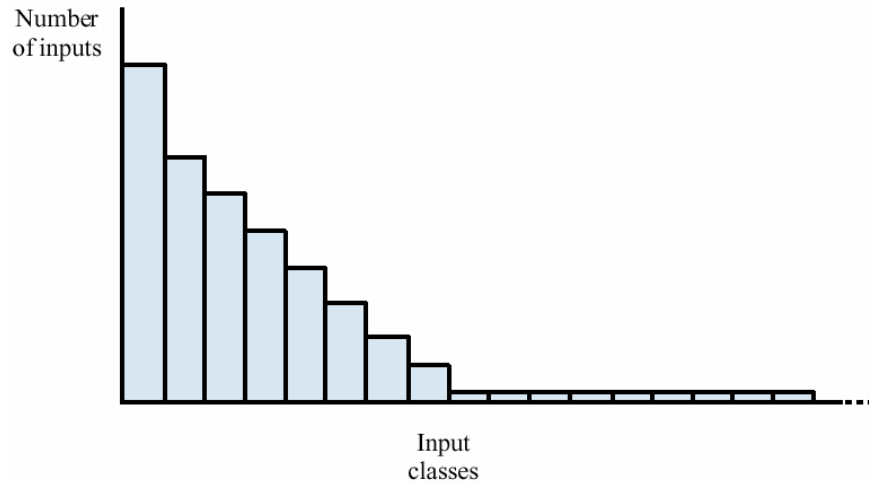
Validazione dell'affidabilità: processo

- **Definire il profilo d'uso (operational profile)**
- **Costruire i test case per il profilo d'uso**
- **Testare, raccogliere numero e istante dei difetti**
- **Calcolare l'affidabilità**
 - Richiede un numero di difetti statisticamente significativo

Operational profile (profilo d'uso)

- **Set di test cases che rispecchia l'effettiva frequenza di input durante l'uso reale**
 - Maggiore la somiglianza, migliore la stima di affidabilità
- **Può essere generato, date ipotesi sull'uso reale**
 - Relativamente semplice per i casi nominali, non semplice per i casi particolari
- **Può provenire da uso di sistema reale (se disponibile)**

Operational profile



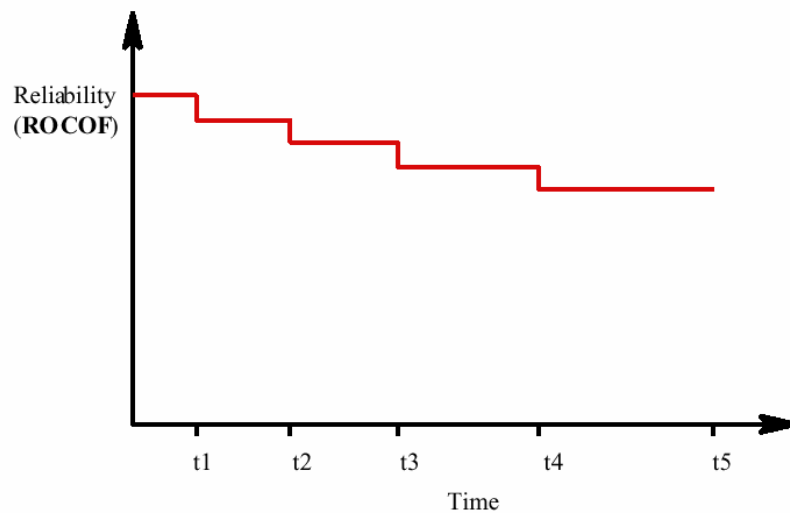
157

Modelli di affidabilita'

- **Modelli matematici di come la affidabilita' cambia (cresce) con il tempo (con la rimozione dei difetti)**
- **Usato per stimare la affidabilita del sistema in uso**

158

Equal-step reliability growth



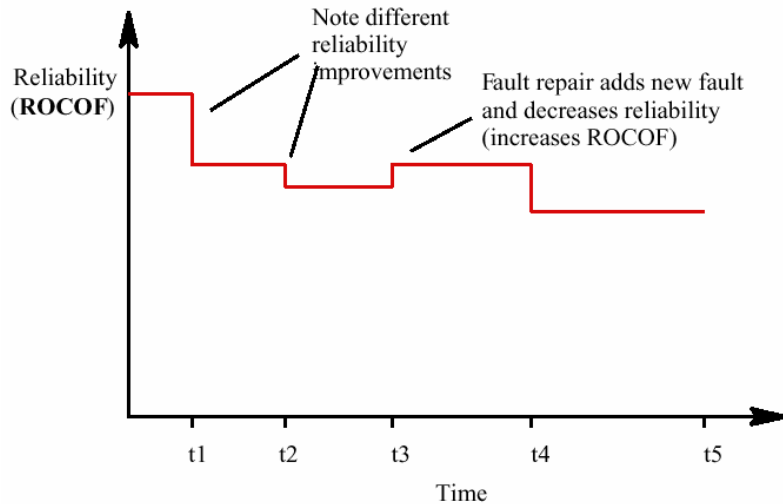
159

Equal step reliability growth

- **Semplice ma non realistico**
 - Riparazione di un fault ne puo' introdurre altri
 - Il trend di miglioramento dell'affidabilita' in genere decresce man mano che I difetti vengono rimossi

160

Random-step reliability growth



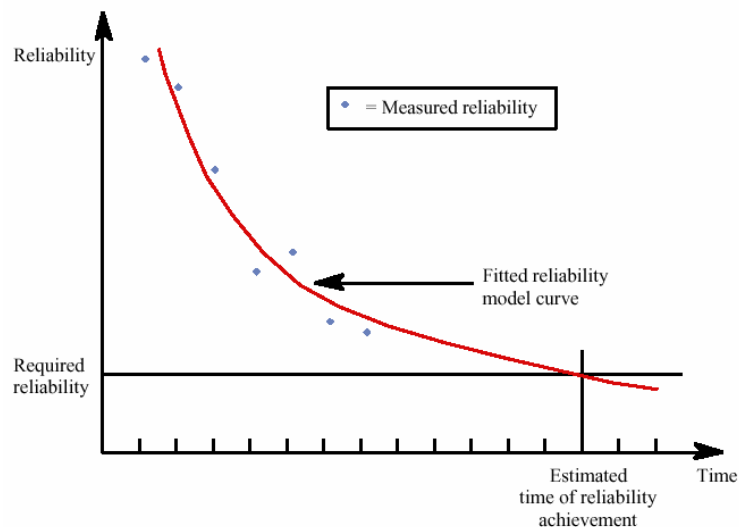
161

Selezione del modello

- **Molti modelli disponibili**
- **Non ne esiste uno migliore in tutti i casi**
- **Provarli tutti e usare il best-fit nel caso in questione**

162

Reliability prediction



163

Problemi nella validazione di affidabilità

- **Incertezza sul profilo d'uso**
 - Riflette l'uso reale?
- **Costo di generazione dei test**
 - L'alto numero di test case da generare rende il processo lungo e costoso
- **Validità statistica, specialmente per sistemi ad alta affidabilità**
 - Il numero di failures generate può essere troppo basso per avere validità statistica

164

Debugging

165

Debugging

- **Testing e QA in generale: rilevazione di difetti (failure + fault)**
 - **Debugging: localizzazione e rimozione di fault**
 - Processo non del tutto 'disciplinato' da metodi assestati; ancora soggetto alla 'arte' di chi lo effettua;
-

166

Debugging

- **Processo difficile implicante una profonda comprensione del codice;**
 - **Condotto ed eseguito in maniera non corretta può inficiare gravemente la qualità del prodotto;**
 - **Alla fine in un processo di debugging si ha una delle due condizioni:**
 - Il fault causa della failure e' stato trovato e rimosso;
 - Il fault non è stato identificato, ma si hanno 'sospetti' su di esso; sono necessari altri opportuni test che validino i sospetti.
-

167

Difficolta'

- Sintomo e causa 'geograficamente' lontani;
- **Sintomo che scompare temporaneamente dopo la correzione di un altro fault;**
 - **Sintomo causato non da un errore specifico (es. arrotondamenti);**
 - **Sintomo causato da un errore umano, non facilmente individuabile (es. stessa variabile con 2 nomi diversi ma simili per errata digitazione);**
 - **Sintomo dovuto a temporizzazione e non ad elaborazione;**
 - **Difficoltà nel riprodurre esattamente gli input che hanno determinato il malfunzionamento (es. Sistemi real time);**
 - **Sintomo intermittente.**
-

168

Difficolta'

• **Importanti anche i fattori psicologici: più aumenta la gravità della failure più aumenta la probabilità di introdurre nuovi fault dovuti ad una affrettata correzione**

• **Il debugging va effettuato sulla base di**

- Valutazioni sistematiche;
- Intuizioni;
- Es. Decomposizione binaria sulla base di ipotesi di lavoro che consentono di individuare i nuovi valori da esaminare.

169

Approcci

- **'forza bruta'**
- **Backtracking**
- **Processo di eliminazione successiva**

170

Forza bruta

Il più comune e, quasi sempre, meno efficiente;

- Viene fatto il dumping della memoria;
- Viene modificato il codice con l'introduzione di sonde che segnalino i cammini eseguiti, ricompilato il programma, rieseguito ed analizzate le stampe (spesso eccessive) delle sonde. L'eccesso di informazioni prodotte può portare al successo ma con gran spreco di risorse.

171

Backtracking

• **Abbastanza comune ed usato con successo per molti programmi;**

- Si parte dal punto in cui si è rilevato il sintomo (generalmente un'istruzione di output) e si procede all'indietro sul codice sorgente (seguendo il flusso di controllo) fino al punto individuante la causa;

• **Programmi con molte linee di codice possono rendere ingestibile il processo.**

172

Eliminazione successiva

- Si basa sull'individuazione/deduzione e decomposizione binaria;
- Si considerano (e organizzano) tutti quei dati che possono essere correlati al sintomo;
- Si effettua un'ipotesi e si utilizzano i dati considerati per validarla o confutarla; oppure
- Si costruisce una lista di tutte le cause possibili e test per eliminare le ipotesi scorrette;
- Si effettua un raffinamento dei dati considerati man mano che le ipotesi convergono sulla causa.

173

Tools

- Compilatori con debugger;
- Tracer;
- Generatori di test case;
- Dumper di memoria;
- Cross reference lister;
- Ma soprattutto la documentazione del sw (completa, aggiornata, consistente, coerente con il codice eseguito).

174

Test Driven Development (TDD) Pair Programming

175

Origini

- Agile methodologies, XP
 - Kent Beck, Extreme Programming Explained.
 - Kent Beck, Test Driven Development by Example.

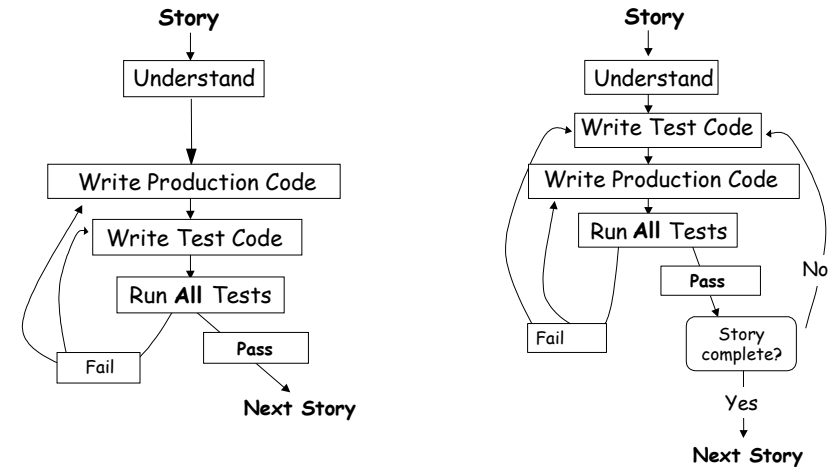
176

Filosofia TDD

- **Test case come (unica) documentazione di un progetto oltre al codice**
- **Test case come codice**
 - Ottima integrazione di unit test e ambiente di sviluppo
 - JUnit e simili
- **Test case scritti prima del codice**

177

Processo tradizionale - TDD



178

Test cases come codice

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

```
testRaddoppia2(){
    assertEquals(4, raddoppia(2));
} // PASS
```

```
testRaddoppia3(){
    assertEquals(6, raddoppia(3));
} // FAIL
```

179

Supporto dei tool: Junit

- **Test framework**
 - plug-in di Eclipse
 - per fare test cases come codice Java
 - framework: set di classi e convenzioni per usarle

```
• Test code
testRaddoppia2(){
}
}
```

```
• Production code
int raddoppia(){
}
}
```

180

Junit

- **É possibile usare JUnit all'interno di Eclipse per eseguire i programmi invece di scrivere delle classi con il metodo `main()`.**
- **É sufficiente:**
 - scrivere una sotto-classe della classe `TestCase`
 - aggiungere ad essa dei metodi di test

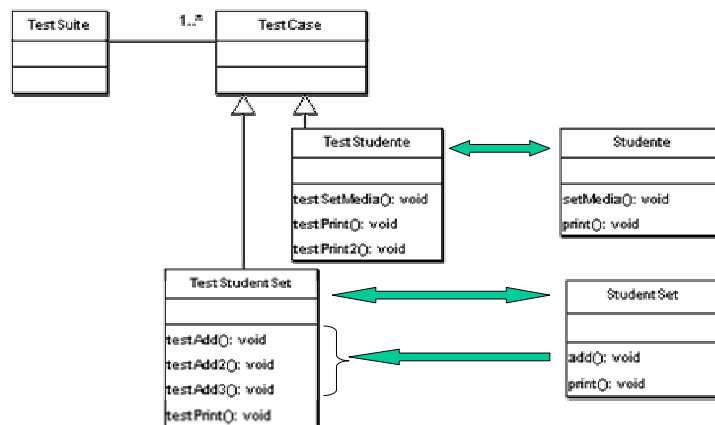
181

Elementi del framework

- **assert*()**
 - funzioni di confronto
- **TestCase**
 - classe contenente serie di test
 - una per ogni classe di production code
 - vari test per ogni metodo di classe production
- **TestSuite**
 - classe contenente serie di TestCase

182

Framework e convenzioni



183

Assert*()

- Metodi definiti su `TestCase`
- **per condizione**
 - `assertTrue("message when test fails", condition);`
- **per valori di ritorno object, int, longs, byte**
 - `assertEquals(expected_value, expression);`
- **per valori di ritorno float, double:**
 - `assertEquals(expected_value, expression, error);`
- **se la condizione testata**
 - e' vera, esegue istruzione seguente
 - e' falsa, break a fine metodo

184

Studente.get/setMediaVoti()

•Test code

```
public class TestStudente extends
TestCase {
public void testGetMediaVoti() {
Studente s = new Studente();
s.setMediaVoti(25.5);
assertEquals(25.5,
s.getMediaVoti(), 0.005);
}
}
```

•Production code

```
public class Studente {
int matricola;
double mediaVoti;
Studente next;

void setMediaVoti(
double voto){
mediaVoti = voto; }
double getMediaVoti() {
return mediaVoti; }
}
```

185

Regole

•Test code

```
public class TestStudente extends TestCase {
public TestStudente(String name){
super(name);
}
public void testGetMediaVoti() {
Studente s = new Studente();
s.setMediaVoti(25.5);
assertEquals(25.5,
s.getMediaVoti(), 0.005);
}
}
```

deve esserci

nome metodo deve
iniziare con "test"

186

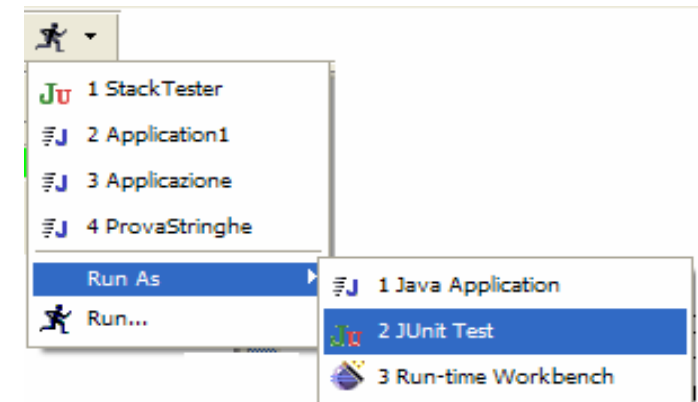
Funzionamento

- **Per un test case JUnit:**
 - Esegue tutti i suoi metodi di test pubblici
 - Ovvero quelli che iniziano con "test"
 - Ignora tutto il resto
- **La classe può contenere metodi di supporto (helper methods)**
 - Non sono pubblici o non iniziano con "test"
 - Possono essere chiamati dai metodi di test

187

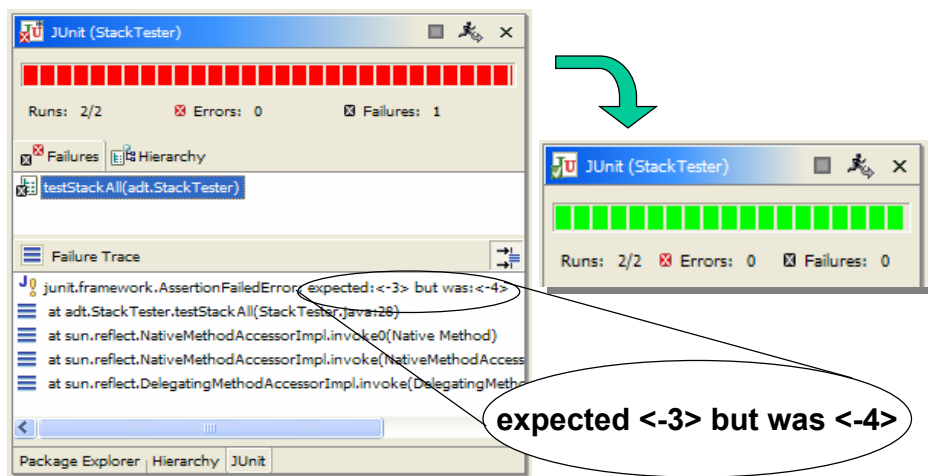
Run as JUnit Test

- Run
- Run As..
- Junit Test



188

Red / Green Bar



189

Pair programming

- “All production code is written with two people looking at one machine”
 - Persona 1: Scrive il codice
 - Persona 2: Pensa in modo strategico a migliorie, test cases, problemi
- Le coppie cambiano in modo continuo
- Vantaggi
 - Conoscenza condivisa del sistema
 - Training on the job
 - Ispezione continua
- Problemi:
 - Produttività
 - Coppie incompatibili

190

Risultati - PP

- **Qualità**
 - Migliore per le coppie
 - Sia con studenti che con professionals
 - Migliora la qualità del programma senza cambiare la qualità del programmatore
 - Effettivo specialmente per programmatori di media esperienza
 - Non difficile da imparare
 - Può essere meno gradito da programmatori esperti
- **Produttività**
 - Peggiora per le coppie
 - In un caso produttività simile – contesto: algoritmi complicati

191

Risultati - TDD

- **Qualità**
 - Migliore per TDD (rispetto Test Last)
- **Produttività**
 - Risultati non chiari
- **Difficile da imparare**

192

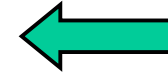
Inspections

193

Tecniche di V&V

•Statiche

- reading (inspections, walkthroughs)
- source code analysis



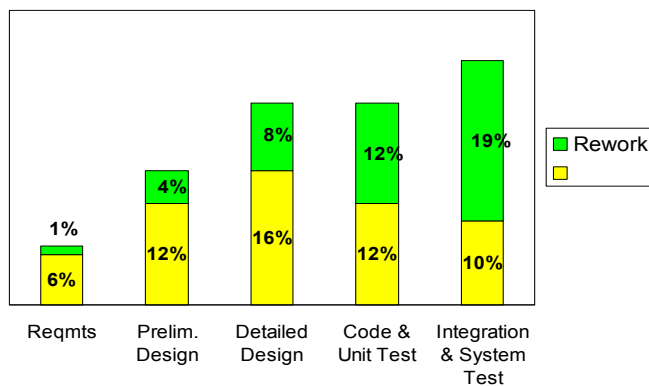
•Dinamiche (esecuzione di codice o modelli)

- testing

194

Rework Problem

- **Avoidable rework accounts for 40-50% of development** [Boehm, 1987; Boehm&Basili, 2001]
 - More recent data available at www.cebase.org



195

Inspection/walkthrough

- **Consiste nella lettura di codice/ documenti da parte di persone, con lo scopo di trovare difetti**
- **Non richiede esecuzione di codice**
 - Si puo' applicare a documenti (requisiti, design, test, ..)
 - Si puo' eseguire prima (anche se sul codice)
- **Efficace**
 - Riusa conoscenza del dominio dei lettori
 - Puo' trovare molti difetti in contemporanea
 - Nel test ci si concentra su un difetto per volta
 - Un difetto puo' mascherarne altri

196

Inspection walkthrough

•Lettura

- Documenti, codice
- Group dynamics (3+ persone)
- Con autore

•Inspection

- Partecipante 'narra' il programma

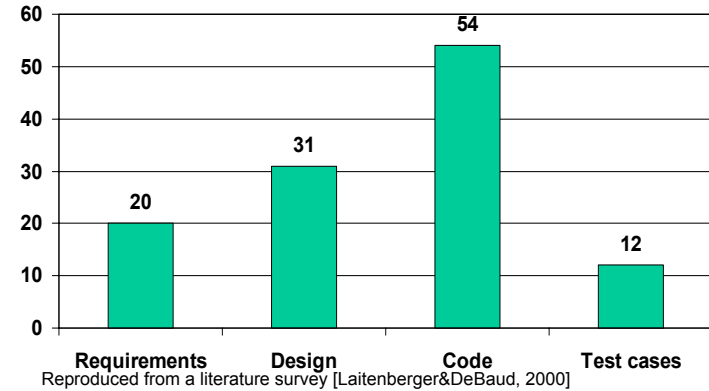
•Walkthrough

- Partecipanti definiscono test case ed 'eseguono' il programma

197

Application of inspection

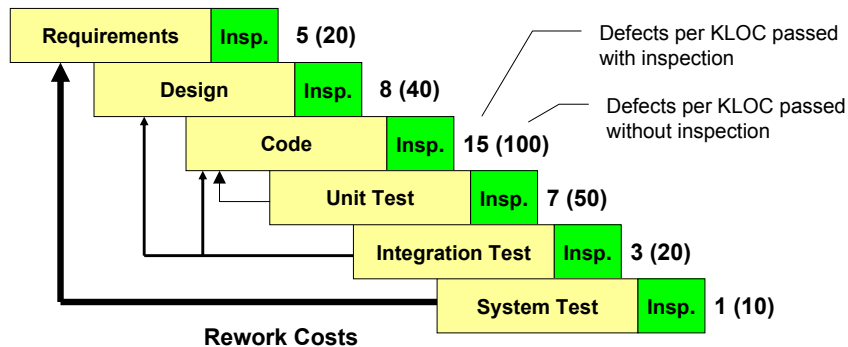
- Can be applied to any life cycle product



198

Benefits

- **Early defect detection improves product quality and reduces avoidable rework (down to 10-20%)**
 - Data from industry averages [Capers Jones, 1991]
 - more data available at www.cebase.org



199

Ispezione e testing

- **Tecniche complementari e non mutuamente esclusive**
 - Usare entrambe nella V and V
- **Ispezione non puo' verificare aspetti non funzionali**
 - Prestazioni, usabilita' ..

200

Ruoli

- **Moderatore:**
 - tipicamente proviene da un altro progetto. Presiede le sedute, sceglie i partecipanti, controlla il processo
- **lettori, addetti al test:**
 - leggono il codice al gruppo, cercano difetti
- **autore:**
 - partecipante passivo; risponde a domande quando richiesto
 - (in alcune varianti e' lettore)
- **Almeno 3-4 partecipanti**

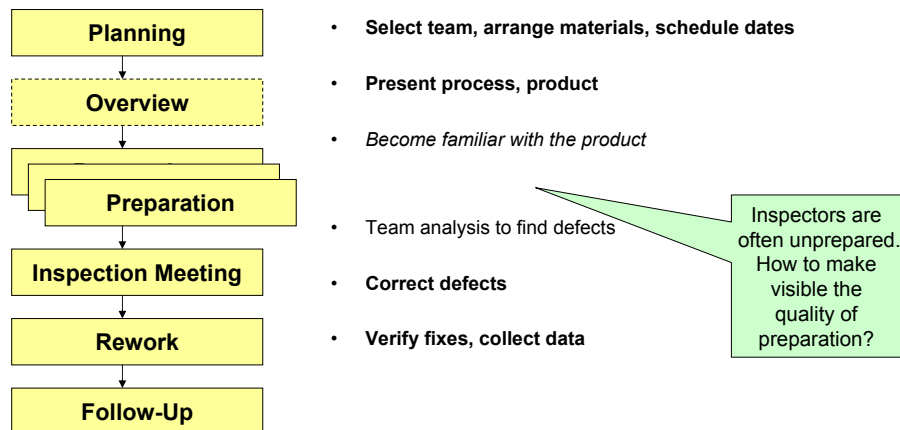
201

Obiettivo

- **Trovare difetti,**
- **NON correggerli**

202

Fagan Inspection Process



203

Processo

- **Presentazione sintetica del sistema al team**
- **Distribuzione al team di codice e altri documenti necessari**
- **Meeting, presa nota di difetti (o presunti difetti)**
 - parafrasare il codice linea per linea, ricostruire l'obiettivo del codice dal sorgente
 - necessario restare in tema
 - seguire le checklist
 - trovare e registrare difetti, ma non correggerli
 - il moderatore è responsabile di evitare anarchia
- **Modifiche da parte di autore**
- **Ripetizione dell'ispezione**
 - Dipende dai casi

204

Entry criteria (Precondizioni all'ispezione (di codice))

- Specifica precisa e completa deve essere disponibile
- Membri del team devono conoscere gli standard dell'organizzazione
- Il codice deve essere sintatticamente corretto
- Checklist deve essere disponibile
- Il management deve accettare il maggior costo (effort) a monte
- Il management non deve usare le ispezioni come forma di valutazione del personale

205

Exit criteria

- All major defects repaired

206

Tempistiche

- 500 istruzioni/ora per la overview
- 125 istruzioni sorgente/ora durante la preparazione individuale
- 90-125 istruzioni/ora durante il meeting
 - Processo costoso sia come effort sia come calendar time (disponibilita' di piu' persone insieme)
 - Ispezione di 500 linee, 40 person hours

207

Preparation

- How to guide individuals in getting familiar with document, and find defects as effectively as possible?
 - Reading techniques for individual analysis
 - Ad-hoc reading
 - Ask inspectors to share a defect taxonomy
 - Checklist-based reading
 - Ask inspectors to answer a categorized series of questions
 - Scenario-based reading
 - Ask inspectors to create an appropriate abstraction
 - » Help to understand the product
 - Ask inspectors to answer a series of questions tailored to the abstraction
- Inspectors follow different scenarios each focusing on specific issues

208

Ad hoc

209

Defect Taxonomies for Requirements

One level

[Basili et al., 1996]

- **Omission**
- **Incorrect Fact**
- **Inconsistency**
- **Ambiguity**
- **Extraneous Information**

Two levels

[Porter et al., 1995]

- **Omission**
 - Missing Functionality
 - Missing Performance
 - Missing Environment
 - Missing Interface
- **Commission**
 - Ambiguous Information
 - Inconsistent Information
 - Incorrect or Extra Functionality
 - Wrong Section

210

Checklists

211

Checklists for Requirements

- **Based on past defect information**
- **Questions refine a defect taxonomy**

[Ackerman et al., 1989]

- **Completeness**
 - 1. Are all sources of input identified?
...
 - 12. For each type of run, is an output value specified for each input value?
...
- **Ambiguity**
 - 18. Are all special terms clearly defined?
...
- **Consistency**
 - ...

212

Checklists for code

- **Con errori piu' comuni**
 - Idealmente la checklist viene adeguata in base agli errori trovati
- **Dipende dal linguaggio di programmazione**

213

Fault class	Inspection check
Data faults	Are all program variables initialised before their values are used? Have all constants been named? Should the lower bound of arrays be 0, 1, or something else? Should the upper bound of arrays be equal to the size of the array or Size -1? If character strings are used, is a delimiter explicitly assigned?
Control faults	For each conditional statement, is the condition correct? Is each loop certain to terminate? Are compound statements correctly bracketed? In case statements, are all possible cases accounted for?
Input/output faults	Are all input variables used? Are all output variables assigned a value before they are output?
Interface faults	Do all function and procedure calls have the correct number of parameters? Do formal and actual parameter types match? Are the parameters in the right order? If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	If a linked structure is modified, have all links been correctly reassigned? If dynamic storage is used, has space been allocated correctly? Is space explicitly de-allocated after it is no longer required?
Exception management faults	Have all possible error conditions been taken into account?

Inspection checks

214

Scenario based

215

Defect-Based Reading

[Porter et al., 1995]

- **A scenario-based reading technique to detect defects in requirements expressed in a formal notation (SCR)**
- **Each scenario focuses on a specific class of defects**
 - data type inconsistencies
 - incorrect functionality
 - ambiguity/missing functionality

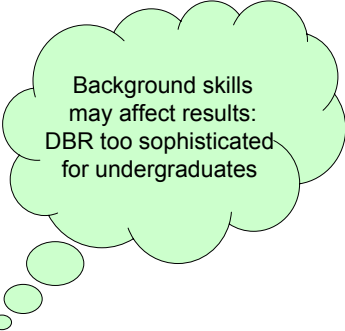
Excerpt from incorrect functionality scenario

1. For each functional requirement identify all input/output data objects:
questions ...
2. For each functional requirement identify all specified system events:
(a) Is the specification of these events consistent with their intended interpretation?
3. Develop an invariant for each system mode:
questions ...

216

Empirical Validation of DBR

- **Hypothesis**
 - Inspections using systematic techniques with specific and distinct responsibilities (DBR) are more effective (find more defects) than those using non-systematic techniques with general and identical responsibilities (ad-hoc and checklist-based reading)
- **Replicated studies within the ISERN community**
 - U. Maryland with graduate students
 - positive evidence
 - Lucent Tech. with practitioners
 - positive evidence
 - U. Bari with undergraduate students
 - no evidence
 - U. Strathclyde with undergraduate students
 - no evidence
 - U. Linköping with undergraduate students
 - no evidence



Background skills may affect results: DBR too sophisticated for undergraduates

217

Perspective-Based Reading

[Basili et al., 1996]

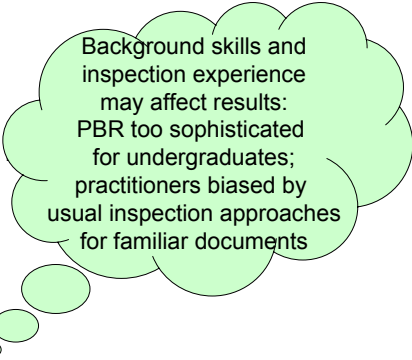
- **A scenario-based reading technique to detect defects in requirements expressed in natural language**
 - extended later for design and source code
- **Each scenario focuses on reviewing the document from the point of view of a specific stakeholder**
 - User (abstraction required: user tasks descriptions)
 - Designer (abstraction required: design)
 - Tester (abstraction required: test suite)

For each requirement/functional specification, generate a test or set of tests that allow you to ensure that an implementation of the system satisfies the requirement/functional specification. Use your standard test approach and technique, and incorporate test criteria in the test suite. In doing so, ask yourself the following questions ...

218

Empirical Validation of PBR

- **Hypothesis**
 - Inspections using systematic techniques with specific and distinct responsibilities (PBR) are more effective (find more defects) than those using non-systematic techniques with general and identical responsibilities (ad-hoc and checklist-based reading)
- **Replicated studies within the ISERN community**
 - U. Maryland with practitioners
 - positive evidence for generic reqmts docs
 - no evidence for NASA-specific reqmts docs
 - U. Kaiserslautern with graduate students
 - positive evidence for generic reqmts docs
 - U. Strathclyde with undergraduate students
 - no evidence for OO code
 - Fraunhofer IESE but no inspection experience
 - positive evidence for OO design docs
 - Fraunhofer IESE at but no inspection experience
 - positive evidence for code



Background skills and inspection experience may affect results: PBR too sophisticated for undergraduates; practitioners biased by usual inspection approaches for familiar documents

219

Fattori umani

- **difetti trovati nell'ispezione non sono usati nella valutazione personale**
 - i programmatori non hanno motivo di nascondere i difetti
- **difetti trovati durante il test (dopo l'ispezione) sono usati nella valutazione personale**
 - i programmatori sono incentivati a trovare difetti nell'ispezione, ma non inserendone intenzionalmente

220

Tools

- automazione di controlli ovvi (per esempio formattazione)
- riferimenti: Checklists, standard con esempi
- mettere a fuoco parti rilevanti (evidenziate automaticamente nel testo)
- annotazioni e comunicazione
- descrizione e (parziale) “forzatura” del processo (esempio negare i diritti di check-off ad un ispettore fino a quando non ha esaminato tutte le parti rilevanti di un processo)

- Supporto a ispezione meeting-less

221

Risultati

•Evidenza empirica sul fatto che inspection e' cost effective. Spiegazioni:

- Processo formale, dettagliato con tracciamento dei risultati
- Check-lists: processo che si automigliora
- Aspetti sociali del processo, specialmente per gli autori
- Considera l'intero spazio di ingresso
- Si applica anche a programmi incompleti

•Limiti

- Scala: tecnica inerentemente a livello di unità
- Non incrementale

222

References

- A.F. Ackerman, L.S. Buchwald, F.H. Lewski. Software inspections: an effective verification process. IEEE Software, 6(3), 1989, 31-36.
- V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sorumgard, and M. Zelkowitz. The empirical investigation of perspective-based reading. Empirical Software Engineering. 1(2), 1996, 133-164.
- V.R. Basili, R. Selby. Comparing the effectiveness of software testing strategies. IEEE Trans. Software Engineering, 13(12), 1987, 1278-1296.
- B. Boehm. Industrial Metrics Top-10 List. IEEE Software, September 1987, 84-85.
- B. Boehm. V.R. Basili. Software Defect Reduction Top-10 List. Computer, January 2001, 3-4.
- B. Boehm, P. Grünbacher, R.O. Briggs. Developing groupware for requirements negotiation: Lessons learned. IEEE Software, 18(3) 2001, 46-55.
- Capers Jones. Applied Software Measurement. McGraw Hill, 1991.
- R.G. Ebenau, S.H. Strauss. Software Inspection Process. McGraw Hill, 1993.
- M.E. Fagan, Design and code inspection to reduce errors in program development. IBM Systems Journal, 15(3), 1976, 182-211.
- D.P. Freedman, G.M. Weinberg. Handbook of Walkthroughs, Inspections, and Technical Reviews. Dorset House, 1990, 89-161.

223

References

- T. Gilb, D. Graham. Software Inspection. Addison-Wesley, 1993.
- W. S. Humphrey. Managing the Software Process. Addison-Wesley, 1989.
- O. Laitenberger, J. DeBaud. An encompassing life cycle centric survey of software inspection. The Journal of Systems and Software 50 (2000), 5-31.
- F. Lanubile, T. Mallardo. Tool support for distributed inspection. Proc. of the 26th Annual International Computer Software & Applications Conference (COMPSAC 2002), Oxford, England, August 2002.
- F. Lanubile, T. Mallardo. Preliminary evaluation of tool-based support for distributed inspection. Proc. of the ICSE Workshop on Global Software Development, 2002.
- F. Lanubile, and T. Mallardo, "An Empirical Study of Web-Based Inspection Meetings", Proc. of the 2nd International Symposium on Empirical Software Engineering (ISESE 2003), Roman Castles (Rome), Italy, IEEE Computer Society Press, September/October 2003.
- National Aeronautics and Space Administration. Software Formal Inspection Guidebook. NASA-GB-A302. Approved on 1993. Available at <http://satc.gsfc.nasa.gov/fi/gdb/fitext.txt>
- National Aeronautics and Space Administration. Software Formal Inspection Process Standard. NASA-STD-2202-93. Approved on 1993. Available at <http://satc.gsfc.nasa.gov/fi/std/fistdtx.txt>

224

References

- D.E. Perry, A. Porter, M.W. Wade, L.G. Votta, and J. Perpich. Reducing Inspection Interval in Large-Scale Software Development. IEEE Transaction on Software Engineering 28, 7 (695-705), 2002.
- A.A Porter, P.M. Johnson. Assessing software review meetings: results of a comparative analysis of two experimental studies. IEEE Trans. Software Engineering, 23(3), 1997, 120-144.
- A.A. Porter, L.G. Votta, V.R. Basili. Comparing detection methods for software requirements inspections: A replicated experiment. IEEE Trans. Software Engineering, 21(6), 1995, 563-575.
- C. Sauer, D.R. Jeffery, L. Land, P. Yetton. The effectiveness of software development technical reviews: A behaviorally motivated program of research. IEEE Trans. Software Engineering, 26(1), 2000, 1-14.
- M. van Genuchten, C. van Dijk, H. Scholten, D. Vogel. Using Group Support Systems for software inspections. IEEE Software, 18(3) 2001, 60-65.
- L.G. Votta. Does every inspection need a meeting? ACM Software Eng. Notes, 18(5), 107-114.


225

Static analysis

226

Tecniche di V&V

•Statiche

- Inspections
- Analisi del codice sorgente 
 - Analisi statica in compilazione
 - » Control flow analysis
 - » Data flow analysis
 - Esecuzione simbolica

•Dinamiche

- testing

227

Analisi statica in compilazione

- I compilatori effettuano una analisi statica del codice per verificare che un programma soddisfi particolari caratteristiche di correttezza statica, per poter generare il codice oggetto.
- Le informazioni e le anomalie che può rilevare un compilatore dipendono dalle caratteristiche del linguaggio e dalle facility di cui esso dispone.
- Es. linguaggi con regole di visibilità statica dei nomi permettono la rilevazione di un maggiore numero di anomalie di quelli con regole di visibilità dinamiche.

228

Analisi statica in compilazione

- La generazione di Cross Reference List risulta molto utile in successive analisi del codice per l'individuazione di anomalie non rilevabili dal compilatore.
- Tipiche anomalie identificabili: nomi di identificatori non dichiarati, incoerenza tra tipi di dati coinvolti in una istruzione, incoerenza tra parametri formali ed effettivi in chiamate a subroutine, codice non raggiungibile dal flusso di controllo
- Molto utili come supporto alle inspections (ma non possono sostituirsi ad esse)

Controlli possibili

Fault class	Static analysis check
Data faults	Variables used before initialisation Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output faults	Variables output twice with no intervening assignment
Interface faults	Parameter type mismatches Parameter number mismatches Non-usage of the results of functions Uncalled functions and procedures
Storage management faults	Unassigned pointers Pointer arithmetic

Passi dell'analisi statica

- **Control flow analysis.** Ricerca cicli con uscite o ingressi multipli, codice non raggiungibile, etc.
- **Data use analysis.** Variabili non inicializzate, var scritte piu' volte, var dichiarate e mai usate
- **Interface analysis.** Controllo tra dichiarazioni (prototipi) e chiamate di procedure.

Passi dell'analisi statica

- **Information flow analysis.** Identifica le dipendenze delle variabili di output. Non trova anomalie ma evidenzia informazioni per inspections
- **Path analysis.** Identifica cammini seguiti nel programma per certo input. Utile per inspections.
 - Entrambe le tecniche producono molta informazione e devono essere usate con attenzione.

138% more lint_ex.c

```
#include <stdio.h>
printarray (Anarray)
int Anarray;
{
  printf("%d",Anarray);
}
main ()
{
  int Anarray[5]; int i; char c;
  printarray (Anarray, i, c);
  printarray (Anarray) ;
}
```

139% cc lint_ex.c
140% lint lint_ex.c

```
lint_ex.c(10): warning: c may be used before set
lint_ex.c(10): warning: i may be used before set
printarray: variable # of args. lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(11)
printf returns value which is always ignored
```

LINT static analysis

233

Analisi statica

- **Piu' utile per linguaggi con weak typing (C)**
- **In molti casi le funzioni di analisi statica sono inglobate dagli ambienti di sviluppo**

234

Esecuzione simbolica

- **Il programma non è eseguito con i valori effettivi ma con valori simbolici dei dati di input**
- **L'esecuzione procede come una esecuzione normale ma non sono elaborati valori bensì formule formate dai valori simbolici degli input**
- **Gli output sono formule dei valori simbolici degli input**
- **L'esecuzione simbolica anche di programmi di modeste dimensioni può risultare molto difficile**
- **Ciò è dovuto all'esecuzione delle istruzioni condizionali: deve essere valutato ciascun caso (vero e falso); in programmi con cicli ciò può portare a situazioni difficilmente gestibili.**

235

Esecuzione simbolica

```
1 function product (x,y,z: integer):
integer;
2 var tmp1, tmp2: integer;
3 begin
4   tmp1 := x*y;
5   tmp2 := y*z;
6   product := tmp1 * tmp2 / y;
7 end;
```

Stm.	x	y	z	tmp1	tmp2	product
1	X	Y	Z	?	?	?
4	X	Y	Z	X*Y	?	?
5	X	Y	Z	X*Y	Y*Z	?
6	X	Y	Z	X*Y	Y*Z	(X*Y)*(Y*Z)/Y

236

Esecuzione simbolica

read (x, y, z);	X, Y, Z
t:=x+y;	T← X+Y
x:=x+t-1;	X← 2*X+Y-1
z:=t*y;	Z← X*Y + Y * Y
y:=y+1;	Y←Y+1
if x>=0 then	X>=0
X<0	
t:=x+y+z;	T← (2+Y)*(X+Y)
T← X+Y	
write (t);	

237

Riferimenti

238

Riferimenti

- Ian Sommerville, Software Engineering, 6th edition, 2000.
- G. Schulmeyer, et al, Handbook of Software Quality Assurance, 1998.
- T. Gilb, Software Inspection, 1993.
- C. Kaner, J. Falk, H. Nguyen, Testing Computer Software, 1999.
- C. Kaner, J. Bach, and B. Pettichord, Lessons Learned in Testing Computer Software, 2001.

239