Progettazione delle prove (della fase di test)

Laura Semini, Ingegneria del Software

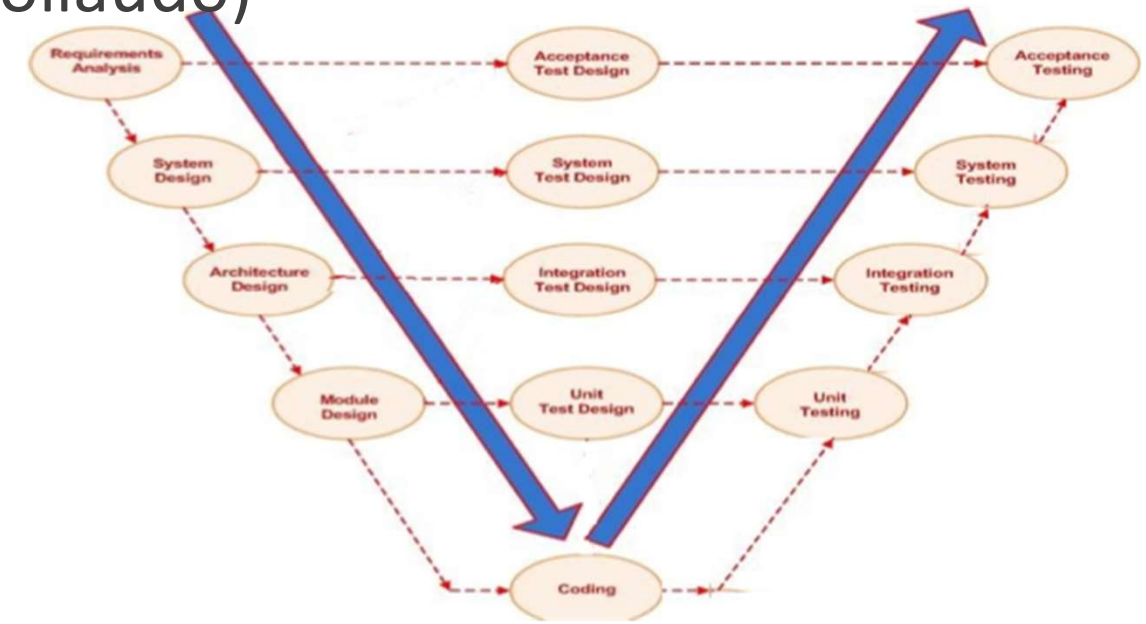
Dipartimento di Informatica, Università di Pisa

Verifica dinamica (testing)

- Si compone di più fasi:
  1. Progettazione
     - input,
     - output atteso,
     - ambiente di esecuzione del test
  2. Esecuzione del codice
  3. Analisi dei risultati
     - output ottenuto con l'esecuzione vs output atteso
  4. Debugging

Test a più livelli

- Test di unità
- Test di integrazione
- Vari tipi di test sul Sistema
- Test di accettazione (o collaudo)



Ripetibilità delle prove

- Bisogna poter ripetere una sessione di test in condizioni immutate
  - Ambiente definito (hardware, condizioni, …)
  - casi di prova definiti (ingressi e comportamenti attesi)
  - procedure definite

Progettazione casi di test

Caso di prova (test case)

- **Un caso di prova** (o test case), è una tripla

<input, output, ambiente>

Test obligation (proof obligation)

- Test obligation = una specifica (descrizione) di casi di test che richiede di testare proprietà ritenute importanti
- Esempio:
  - specifica del test case:
    - "un input formato da due parole e un input formato da tre"
  - i seguenti casi di test sono 2 tra i tanti casi di test che soddisfano la test obligation
    - "alpha beta"
    - "Milano Pisa Roma"

Adeguatezza dei casi di test

- Un modo per definire l'adeguatezza di un insieme di casi di test (test suite)

- Non possiamo dire:
  - Se il sistema supera un'adeguato insieme di test *allora deve essere corretto*
  - Perché la correttezza è indecidibile

Criteri Pratici di (in)Adeguatezza

- Criteri che identificano inadeguatezza nei casi di test
- Esempi:
  - nella specifica del sistema ho due casi, il casi di test non testano che i due casi siano trattati differentemente
  - nel codice ci sono n istruzioni , i casi di test ne testano k<n
- Il fatto che il nostro insieme di test non verifichi un criterio ci suggerisce come modificarlo
- Un criterio di adeguatezza =insieme di test obligations

Soddisfare un criterio di adeguatezza

- Un criterio di adeguatezza = insieme di test obligations
- Un insieme di test soddisfa un criterio di adeguatezza se:
  1. Tutti i test hanno successo
  2. Ogni test obligation è soddisfatta da almeno un caso di test (nell'insieme di casi di test scelto)

- Esempio: il criterio di adeguatezza della copertura delle istruzioni è soddisfatto da un insieme di test S per il programma P se
  1. ogni istruzione eseguibile in P è esercitata da almeno un test in S,
  2. il risultato dell'esecuzione corrisponde a quello atteso.

Come definire le test obligations

- dalle funzionalità (a scatola chiusa, black box), guardando la specifica del sw
  - basati sulla conoscenza delle funzionalità
  - mirati a evidenziare malfunzionamenti relativi a funzionalità
    - Es: il metodo che restituisce il valore assoluto di una variabile in input: le test obligations richiedono di testare il metodo almeno con un valore positivo e almeno con uno negativo
- dalla struttura (a scatola aperta, white box): guardando il codice
  - basati sulla conoscenza del codice
  - mirati a esercitare il codice indipendentemente dalle funzionalità
    - Es: passare da ogni loop almeno una volta

Come definire le test obligations

- dal modello del programma
  - Modelli utilizzati nella specifica o nella progettazione, o derivati dal codice
    - Es: Esercizio di tutte le transizioni nel modello di protocollo di comunicazione
- da fault ipotetici
  - Cercano difetti ipotizzati (bug comuni)
    - Es: check per la gestione del buffer overflow testando con input molto grandi

Progettazione di casi di test: Black-box vs White-box

```
public boolean isMCD(int a, int b, int mcd)
```

- Criteri Black-box test case: basati solo sulla specifica del metodo: *isMCD restituisce true se mcd è il massimo comune divisore di a e b*
- White-box test case basati sul codice

```
public boolean isMCD(int a, int b, int mcd) {
if (a<=0 || b<=0)
      throw new IllegalParameter();
int min =a<b ? a : b; 5)
if ( (a % mcd != 0) | (b % mcd != 0) )
      return false;
for (int i = mcd + 1; i < =min; i++){
         if ( (a % i == 0) && (b % i == 0) )
                  return false; }
return true ;  }
```

Uno sguardo alle altre fasi del testing

Gli elementi di una prova: batteria e procedura

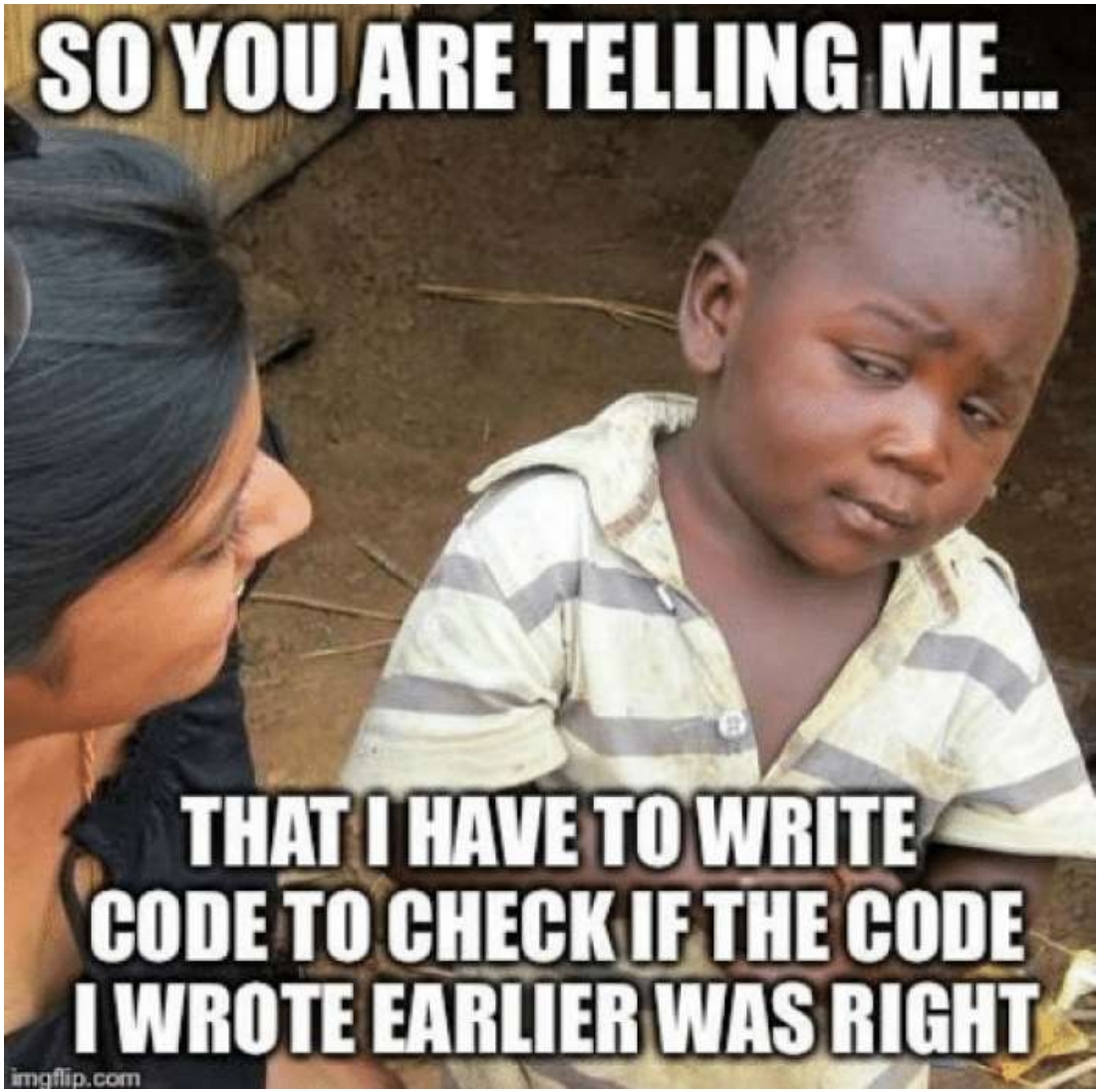
- Batteria di prove (o test suite)
  - un insieme (una sequenza) di casi di prova
- Procedura di prova
  - le procedure (automatiche e non) per eseguire, registrare analizzare e valutare i risultati di una batteria di prove

Procedura di prova

- Definizione dell'obiettivo della prova
- Progettazione della prova
  - scelta e definizione dei casi di prova (della batteria di prove)
- Realizzazione dell'ambiente di prova
  - ci sono driver e stub da realizzare, ambienti da controllare, strumenti per la registrazione dei dati da realizzare
- Esecuzione della prova
  - l'esecuzione può richiedere tempo
- Analisi dei risultati
  - l'esame dei risultati alla ricerca di evidenza di malfunzionamenti
- Valutazione della prova

Realizzazione dell'ambiente di prova: test scaffolding

- Codice aggiuntivo necessario per eseguire un test.
- Si chiama **scaffolding** (impalcatura), per analogia alle strutture temporanee erette attorno a un edificio durante la costruzione o la manutenzione.



Test Scaffolding

- Lo scaffolding può includere:
  - **driver di test** (sostituiscono un programma principale o di chiamata),
  - **stub** (sostituiscono funzionalità chiamate o utilizzate dal software in prova) (**mock**),
    - **se il metodo A usa B che usa C, per testare B devo costruire del codice che simula A (driver) e del codice che simula C (stub)**
  - **test harness** (sostituiscono parti dell'ambiente di distribuzione) (ATTENZIONE: per altri autori harness è sinonimo di scaffolding)
  - **tool per gestire l'esecuzione del test**
  - **tool per registrare i risultati**

Ripasso

- Verifica vs validazione
- Verifica è indecidibile
- Verifica statica vs dinamica (aka testing)
- Caso di test <input, output atteso, ambiente di esec.>
- Scaffolding, fa parte ambiente esecuzione:
  - driver,
  - stub (aka mock),
  - test harness (ambiente di distribuzione)
- Proof obligation (specifica) di casi di test
- Casi di test: valori
- Malfunzionamento vs difetto (aka baco, bug)

Metodi black box per generare valori di input

Sono criteri per l'individuazione dei casi di input che si basano sulle specifiche

Sinonimi: criteri funzionali, a scatola chiusa

Strategia

- Separare le funzionalità da testare
  - Per esempio usando i casi d'uso
- Derivare un insieme di casi di test per ogni funzionalità
- M(p1, p2, p3, p4) < <i1,i2,i3,i4>, output atteso, ambiente>
  - Per fare ciò
    - per ogni (tipo di) parametro di input
      - Si individuano dei valori da testare
      - Per questo si usano alcune tecniche (metodi) che vediamo nei prossimi lucidi
    - Per l'insieme dei parametri
      - Si usano tecniche che vanno sotto il nome di testing combinatorio per ridurre le combinazioni

Metodo statistico

- I casi di test sono selezionati in base alla distribuzione di probabilità dei dati di ingresso del programma
  - Il test è quindi progettato per esercitare il programma sui valori di ingresso più probabili per il suo utilizzo a regime
  - Il vantaggio è che, nota la distribuzione di probabilità, la generazione dei dati di test è facilmente automatizzabile
  - Non sempre corrisponde alle effettive condizioni d'utilizzo del software
  - È oneroso calcolare il risultato atteso (problema dell'oracolo)

Metodo statistico: esempio

- Si consideri l'input "età il giorno della laurea":
  - Il tipo è int
- In questo caso è ragionevole usare il metodo statistico e dare le specifica di test (Test obligation):
  - tutti i valori compresi tra 20 e 27
  - Il 40% dei valori tra 27 e 35
    - Questi possono essere scelti in modo random
  - Il 5% dei valori tra 36 e 100
    - Questi possono essere scelti in modo random
- Casi di test che soddisfano le test obligations:
- <20, _, _>, <21, _, _>,…. <27, _, _>, <29, _, _>,…. <51, _, _>,….
  (al momento non sono ancora specificati output atteso e ambiente)

Partizione dei dati in ingresso in classi di equivalenza(categorie)

- Metodo in cui il dominio dei dati di ingresso è ripartito in **classi di equivalenza** (**categories** nel libro Pezzé Young)
  - due valori d'ingresso appartengono alla stessa classe di equivalenza se, in base ai requisiti, dovrebbero produrre lo stesso **comportamento** del programma (non necessariamente stesso output)

Partizione dei dati in ingresso: esempio

- metodo `int calcolaTasse(int reddito)`
- Test obligation: un caso di test per aliquota
- Casi di test che soddisfano le test obligations:
- <10.000, 2300, _>, <20.000, 4800, _>,….

Scaglioni di reddito	*Aliquote*
Fino a € 15.000	23%
Oltre a € 15.000 e fino a € 28.000	27%
Oltre a € 28.000 e fino a € 55.000	38%
Oltre a € 55.000 e fino a € 75.000	41%
Oltre a € 75.000	43%

Partizione dei dati in ingresso in classi di equivalenza(categorie)

- Il criterio è economicamente valido solo per quei programmi per cui il numero dei possibili comportamenti è sensibilmente inferiore alle possibili configurazioni d'ingresso
  - per come sono costruite le classi, i risultati attesi dal test sono noti e quindi non si pone il problema dell'oracolo

- Il criterio è basato su un'affermazione generalmente plausibile, ma non vera in assoluto
  - la deduzione che il corretto funzionamento sul valore rappresentante implichi la correttezza su tutta la classe di equivalenza dipende dalla realizzazione del programma e non è verificabile sulla base delle sole specifiche funzionali

Valori limite (di frontiera)

- Metodo basato sull'individuazione di valori estremi
  - Estremi delle classi di equivalenza definite in base all'eguaglianza del comportamento del programma
  - Estremi in base a considerazioni inerenti il tipo dei valori d'ingresso (per esempio se interi considerare 0 e 1)

Esempio: frontiera

- metodo `int calcolaTasse(int reddito)`
- Test obligation: provare tutti gli intorni degli estremi degli intervalli
- Casi di test che soddisfano le test obligations:
- <14.990, 3.447,7, _> <15.000, 3450, _>, <15.010, 3452,7 ,_> ….
- (Per questa specifica poco significativo questo criterio:
  - sui punti di frontiera non è derivabile ma è comunque continua)

Scaglioni di reddito	*Aliquote*
Fino a € 15.000	23%
Oltre a € 15.000 e fino a € 28.000	27%
Oltre a € 28.000 e fino a € 55.000	38%
Oltre a € 55.000 e fino a € 75.000	41%
Oltre a € 75.000	43%

Esempio più significativo

- **frontiera punto di discontinuità**
- metodo `int calcolaSconto(int spesa)`

- Casi di test che soddisfano le test obligations:
- <48.99 , 0, _>, <49.00, 7.35, _>, <49.01 , 7.3515 ,_> ….

Valori limite (di frontiera)

- Questo criterio ricorda i controlli sui valori limite tradizionali in altre discipline ingegneristiche per le quali è vera la proprietà del comportamento continuo
  - in meccanica, ad esempio, una parte provata per un certo carico resiste con certezza a tutti i carichi inferiori

- **In realtà** si guardano i valori limite perché spesso è nell'intorno dei valori limite che **si nascondono difetti del codice**

I casi non validi

- Per ogni input si definiscono anche i casi non validi (che devono generare un errore):
  - Età inferiori a 20 o superiori a 120 per la laurea
  - Reddito negativo per il calcolo delle aliquote
  - Spesa negativa per il calcolo dello sconto

Metodo random

- Generare in modo automatico un insieme grande a piacere di valori
  - Costo zero la generazione
  - Non ripetibile e può essere difficile trovare l'output atteso
  - Applicabile se costa poco l'esecuzione
  - Difficilmente considera i casi limite
    - Esempio: trovare le radici di un'equazione di secondo grado

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

    - Quasi impossibile che il caso b=0, a=0 sia generato in modo casuale

Test basato su catalogo

- Nel tempo un organizzazione può essersi costruita un'esperienza nel definire casi di test
- Collezionare questa esperienza in un catalogo può rendere più veloce il processo e automatizzare alcune decisioni riducendo l'errore umano
- I cataloghi catturano l'esperienza di coloro che definiscono i test elencando tutti i casi che devono essere considerati per ciascun possibile tipo di variabile

Esempio di voce nel catalogo

- Assumiamo che una funzione usi una variabile il cui valore deve appartenere ad un intervallo di interi, il catalogo potrebbe indicare i casi seguenti come rilevanti:

1. The element immediately preceding the lower bound of the interval
2. The lower bound of the interval
3. A non-boundary element within the interval
4. The upper bound of the interval
5. The element immediately following the upper bound

- Di fatto, si stanno considerando:
  - l'intervallo in cui è definita la funzione come se fosse un'unica classe di equivalenza
  - la sua frontiera
  - valori non validi

Testing Combinatorio

foo (x1, x2, x3, …)

Tecnica da applicare al crescere del numero dei parametri in input

Esplosione combinatoria

- In presenza di più dati di input, se si prende il prodotto cartesiano dei casi di test individuati, facilmente si ottegono numeri non gestibili
- Occorrono strategie per generare casi di test significativi in modo sistematico
- **Tecniche** per ridurre l'esplosione combinatoria
  - **Vincoli**
  - **Pairwise testing**

Vincoli

- Immaginiamo 5 parametri di input: <x1, x2, x3, x4, x5>
  - Dominio di x1 e x2 ripartibile in 8 classi (di cui una di valori non validi → errore)
  - Dominio di x3 e x5 ripartibile in 4 classi (di cui una di valori non validi → errore)
  - Dominio di x4 ripartibile in 7 classi (di cui una di valori non validi → errore)
- Se prendiamo un rappresentante per classe:

  8*8*4*7*4= 7.168 casi di test

Vincoli

- Si considerano 3 strategie "a vincoli" per ridurre il numero di possibili combinazioni:
  - di errore,
  - di proprietà,
  - singoletti

Vincoli di errore (error constraints)

- <x1, x2, x3, x4, x5>, come prima
- Un rappresentante per classe: 8*8*4*7*4= 7.168 casi di test
- **Viene preso un solo caso, per ogni posizione, con input non valido**
- 5 + 7*7*3*6*3= 2.651
- Da 7.168 a 2.651

Vincoli: property/if property

- Abbiamo ridotto a: 5 + 7 * 7 * 3 * 6 * 3 = 2.651
- Definiamo dei vincoli property/if property sui primi due parametri
  - x1:

    classe 1, classe 2, classe 3, classe 4 [property negativi]
    classe 5, classe 6, classe 7 [property positivi]
    (classe8 [error])
  - x2:

    classe 1, classe 3, classe 5, classe 7 [if negativi]
    classe 2, classe 4, classe 6 [if positivi]
    (classe8 [error])
- 5 + (4*4+3*3) * 3 * 6 * 3= 5 + 1350 = 1.355
- da 7.168 a 2.651 a 1.355

Vincoli: single

- Fino ad ora: 5 + (4*4+3*3) * 3 * 6 * 3
- Per uno (o più) parametri si può decidere di testare un solo valore
- Per esempio applichiamo il metodo "single" ad x4
- 5 + (4*4+3*3) * 3 * 1 * 3= 5 + 225 = 230
- da 7.168 a 2.651 a 1.355 a 230

- (rispetto a error: qui si fissa quel parametro, in error gli altri)

Tecnica basata su vincoli: discussione

- La tecnica basata su vincoli permette di introdurre vincoli che limitino il numero di test ottenuti dalla generazione di tutte le le combinazioni di valori possibili.

- Funziona bene se i vincoli che imponiamo sono **reali vincoli del dominio** e non se li aggiungiamo al solo scopo di limitare le combinazioni

Pairwise testing
Combinazione di test basato su coppie

- foo(x1, x2, x3, x4, x5)

- Nel caso in cui il dominio non contenga in sé dei vincoli è preferibile optare per un'altra tecnica:
  - **la generazione di tutte le combinazioni solo per i sottoinsiemi di 2 variabili**

In generale si possono generare tutte le combinazioni solo per i sottoinsiemi di k variabili, con k<n (pairwise quando k=2)

Combinazione di test basato su coppie

- L'idea: generare tutte le possibili combinazioni solo per le possibili coppie di variabili
- Riprendiamo il nostro esempio: <x1, x2, x3, x4, x5>
- Tutte le combinazioni: 8*8*4*7*4= 7.168
- Quanto si risparmia con il pairwise?
  - 8*8+ 8*4+8*7+8*4 + 8*4+8*7+8*4 + 4*7+4*4 + 7*4

    = 376, in realtà sono anche meno se generiamo le

    combinazioni in maniera efficiente

Esempio

Display Mode
- full-graphics
- text-only
- limited-bandwidth

Language
- English
- French
- Spanish
- Portuguese

Fonts
- Minimal
- Standard
- Document-loaded

Color
- Monochrome
- Color-map
- 16-bit
- True-color

Screen size
- Hand-held
- Laptop
- Full-size

Text-only, Laptop, Standard

Esempio

- Se volessimo generare tutte le combinazioni per *Display mode*, *Screen size* e *Fonts* avremmo

$$3^3 = 27$$

- Generiamo tutte le combinazioni per la coppia <*Display mode*, *Screen size*> abbiamo

$$3^2 = 9$$

- Poi occorre generare anche tutte le combinazioni per le coppie FontsxScreen size e FontsxDispay

- Ma in questo caso generando le combinazioni per la prima coppia il valore del terzo parametro può essere aggiunto in modo da coprire anche tutte le combinazioni di *FontsxScreen size* e *FontsxDispay mode*

Esempio

Display mode × *Screen size*		*Fonts*
Full-graphics	Hand-held	Minimal
Full-graphics	Laptop	Standard
Full-graphics	Full-size	Document-loaded
Text-only	Hand-held	Standard
Text-only	Laptop	Document-loaded
Text-only	Full-size	Minimal
Limited-bandwidth	Hand-held	Document-loaded
Limited-bandwidth	Laptop	Minimal
Limited-bandwidth	Full-size	Standard

- La generazione di combinazioni che in maniera efficiente coprano tutte le coppie è impossibile da fare a mano per molti parametri con molti valori ma può essere fatta con euristiche.

Syllabus

- Cap 9-10-11
  - Software Testing and Analysis: Process, Principles and Techniques-
  - Mauro Pezzè e Michal Young