# 19. Testing

**IS 2024-2025**


**Laura Semini, Jacopo Soldani**
Corso di Laurea in Informatica
Dipartimento di Informatica, Università of Pisa

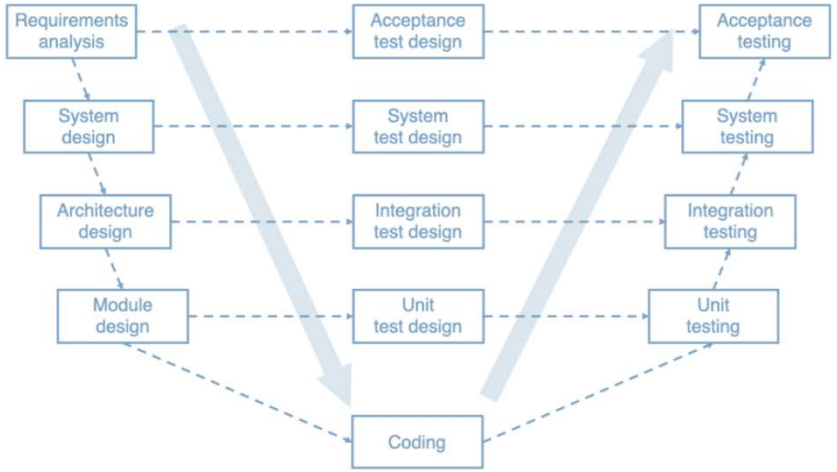
# VERIFICA DINAMICA (AKA. TESTING)

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 vs output atteso)
4. **Debugging**

**Livelli diversi** testati diversamente

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



# PROGETTARE CASI DI TEST

Bisogna poter **ripetere** una sessione di **test** in **condizioni immutate**

- ambiente definito (hardware, condizioni, …)
- casi di prova definiti (input predefiniti e comportamenti/output attesi)
- procedure definite

Un **caso di prova** è una tripla $\langle input, output\_atteso, ambiente \rangle$

# COSA SERVE?

- **Test obligation** = specifica di casi di test utili a verificare proprietà ritenute importanti

  Esempio di specifica: «un input formato da due o tre parole»

  Due casi di test (tra i tanti) che soddisfano la test obligation
  - `alpha beta`
  - `Firenze Pisa Livorno`

- Criteri per definire l'**adeguatezza** di un insieme di casi di test (test suite)

  Esempi:
  - «nella specifica del sistema ho due casi da trattare differentemente, ma i casi di test non testano che i due casi siano trattati differentemente»
  - «nel codice ci sono $n$ istruzioni, ma i casi di test ne testano $k < n$»

- Un **criterio** di adeguatezza è sostanzialmente un **insieme** di **test obligation**

# SODDISFARE UN CRITERIO DI ADEGUATEZZA

Un insieme di test **soddisfa** un criterio di adeguatezza se:

- Tutti i test hanno **successo**
- Ogni **test obligation** è soddisfatta da **almeno un caso di test** (nell'insieme di casi scelti)

Esempio: «tutte le istruzioni del programma sono coperte dai test»

Un insieme di test S soddisfa il criterio scritto sopra per un programma P se

- ogni istruzione eseguibile in P è eseguita da almeno un test in S e
- il risultato dell'esecuzione corrisponde a quello atteso

Se un insieme di test **non verifica** un criterio di adeguatezza, questo ci suggerisce **come modificare** tale insieme di test

# COME DEFINIRE LE TEST OBLIGATION?

- A partire dalle **funzionalità** indicate nella specifica del software (a scatola chiusa, black box)
  - Per evidenziare malfunzionamenti
  - Esempio: «usare almeno un valore positivo e uno negativo» (per un metodo che restituisce il valore assoluto di una variabile)

- A partire dalla **struttura**, guardando il codice (a scatola aperta, white box)
  - Per esercitare il codice indipendentemente dalle funzionalità
  - Esempio: «eseguire ogni loop almeno una volta»

- A partire dal **modello** del programma
  - Modelli utilizzati nella specifica o nella progettazione, o derivati dal codice
  - Esempio: «provare tutte le transizioni nel modello di un protocollo di comunicazione»

- A partire da **fault ipotetici**
  - Cercando difetti ipotizzati o bug comuni
  - Esempio: «verificare la gestione del buffer overflow usando input molto grandi»

# BLACK-BOX VS WHITE-BOX

I criteri **black-box** si basano solo solo sulla specifica del metodo

`isMCD(a,b,mcd)` restituisce `true` se `mcd` è il massimo comune divisore di `a` e `b`

I criteri **white-box** si basano sul codice che implementa il metodo

```
public boolean isMCD(int a, int b, int mcd) {
  if (a<=0 || b<=0)
    throw new IllegalParameter();
  int min = a<b ? a : b;
  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;
}
```

# BATTERIE E PROCEDURE DI PROVA

- **Batteria** di prove, aka **test suite** = un insieme (o sequenza) di casi di prova

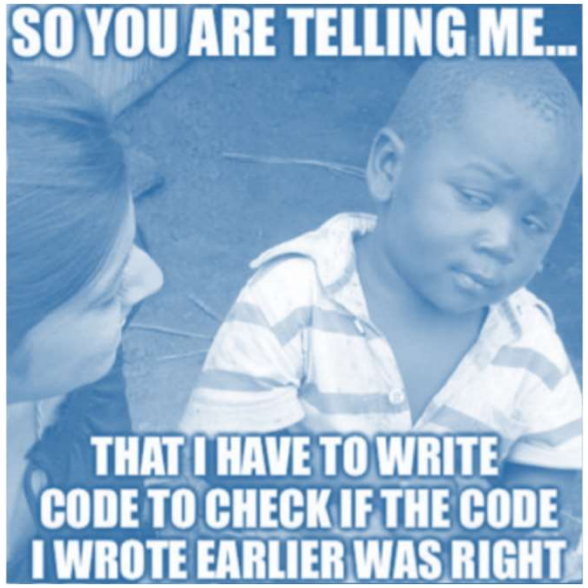
- **Procedura** di prova = procedure (automatiche e non) per eseguire, registrare analizzare e valutare i risultati di una batteria di prove

La procedura di prova si articola in più **fasi**

1. Definizione dell'obiettivo della prova
2. Progettazione della prova (scelta e definizione dei casi di prova, ovvero della batteria di prove)
3. Realizzazione dell'ambiente di prova (driver e stub da realizzare, ambienti da controllare, strumenti per la registrazione dei dati da realizzare)
4. Esecuzione della prova (può richiedere tempo)
5. Analisi dei risultati (alla ricerca di evidenza di malfunzionamenti)
6. Valutazione della prova

# BATTERIE E PROCEDURE DI PROVA (CONT.)

Per eseguire un test, serve codice aggiuntivo, chiamato test **scaffolding** (per analogia con le impalcature temporanee erette attorno a un edificio durante la sua costruzione o manutenzione)




Lo **scaffolding** può includere

- **driver** di test (sostituiscono il chiamante)
- **stub** o **mock** (sostituiscono funzionalità chiamate)

  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)

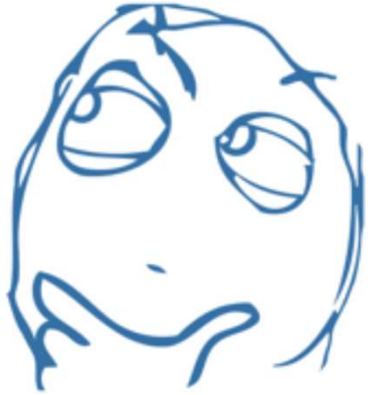
- **test harness**[1] (sostituiscono parti dell'ambiente di distribuzione)
- **tool** per gestire l'esecuzione del test e/o registrarne i risultati

1. In alcuni casi, «harness» è usato come sinonimo di «scaffolding»

# RIEPILOGO

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

**Ok, ma come si generano i test?**



# METODI BLACK-BOX

Generazione dei casi (e valori) di **input** basata sulle **specifiche**
(anche detti criteri funzionali o a scatola chiusa)

**Strategia**

- Separare le **funzionalità da testare** (es. basandosi sui casi d'uso)
- Derivare un insieme di **casi di test** per ogni **funzionalità** (es. per `m(p1, p2, p3, p4)`, serve $\langle\langle i_1, i_2, i_3, i_4\rangle, output\ atteso, ambiente\rangle$)
  - Per ogni (tipo di) **parametro** di input, si individuano dei **valori da testare**
    (vedremo alcune tecniche nei prossimi lucidi)
  - Per l'**insieme dei parametri**, si usano tecniche di **testing combinatorio**

# SELEZIONE VALORI: METODO STATISTICO

Casi di test selezionati in base alla **distribuzione di probabilità** degli input del programma

- Test progettato per esercitare il programma sugli **input più probabili** «a regime»
- Nota la distribuzione di probabilità, la generazione dei dati di test è **automatizzabile** (ma non sempre corrisponde alle effettive condizioni d'utilizzo del software)
- Difficile e oneroso calcolare il risultato atteso (**problema dell'oracolo**)

Esempio: Selezionare i valori per un input che rappresenta «età il giorno della laurea» (tipo int)

- **Test obligation** generate con il metodo statistico
  - tutti i valori compresi tra 20 e 27
  - Il 40% dei valori tra 27 e 35, scelti in modo casuale
  - Il 5% dei valori tra 36 e 100, scelti in modo casuale
- **Casi di test** che soddisfano le test obligation
  - $\langle 20,\cdot,\cdot\rangle, \langle 21,\cdot,\cdot\rangle, \dots, \langle 27,\cdot,\cdot\rangle, \langle 29,\cdot,\cdot\rangle, \dots, \langle 51,\cdot,\cdot\rangle, \dots$
  - **Non specificano** ancora **output atteso** e **ambiente**

# SELEZIONE VALORI: PARTIZIONE IN CATEGORIE

Il dominio degli input è ripartito in **categorie** (aka. **classi di equivalenza**)

- due input appartengono alla **stessa classe di equivalenza** se, in base ai requisiti, dovrebbero produrre lo **stesso comportamento** del programma (non necessariamente stesso output)
- economicamente valido solo se il numero dei **possibili comportamenti** è sensibilmente **inferiore** alle possibili **configurazioni d'ingresso** (ovvero se le classi di equivalenza sono tali per cui i risultati attesi dal test sono noti e non si pone il problema dell'oracolo)
- basato su un'affermazione generalmente **plausibile**, ma non vera in assoluto (il corretto funzionamento dedotto dal singolo valore rappresentante dipende dalla realizzazione del programma e non è verificabile sulla base delle sole specifiche funzionali)

Esempio: Consideriamo `int calcolaTasse(int reddito)`

- **Test obligation** generata con partizione in categorie
  - Un caso di test per aliquota
- **Casi di test** che soddisfano la test obligation
  - $\langle 10.000, 2.300, \cdot \rangle, \langle 20.000, 4.800, \cdot \rangle$, ...

| Scaglioni di reddito | Aliquote |
|---|---|
| Fino a € 15.000 | 23% |
| Da € 15.000 a € 28.000 | 27% |
| Da € 28.000 a € 55.000 | 38% |
| Da € 55.000 a € 75.000 | 41% |
| Oltre € 75.000 | 43% |

# SELEZIONE VALORI: FRONTIERA

Metodo basato sull'individuazione di **valori limite**/estremi

- rispetto alle classi di equivalenza basate sull'uguaglianza di comportamento del programma
- rispetto al tipo degli input (es. se indici di un array di $N$ elementi, considerare $0$ e $N-1$)
- ricorda i controlli sui valori limite tradizionali in altre discipline ingegneristiche per le quali è vera la proprietà del comportamento continuo (es. in meccanica, una parte provata per un certo carico resiste con certezza a tutti i carichi inferiori)
- si guardano i valori limite perché spesso è nell'intorno dei valori limite che si nascondono difetti del codice (e questo si impara naturalmente con l'esperienza)

| Scaglioni di reddito | Aliquote |
|---|---|
| Fino a € 15.000 | 23% |
| Da € 15.000 a € 28.000 | 27% |
| Da € 28.000 a € 55.000 | 38% |
| Da € 55.000 a € 75.000 | 41% |
| Oltre € 75.000 | 43% |

Esempio: Consideriamo `int calcolaTasse(int reddito)`

- **Test obligation** generata con metodo della frontiera
  - Provare tutti gli intorni degli estremi degli intervalli
- **Casi di test** che soddisfano la test obligation
  - $\langle 14.990, 3.448, \cdot \rangle, \langle 15.000, 3.450, \cdot \rangle, \langle 15.010, 3453, \cdot \rangle$, ...
  - Non molto significativi per il caso considerato -.-

# SELEZIONE VALORI: FRONTIERA (CONT.)

Il metodo della frontiera è più significativo quando la frontiera è un **punto di discontinuità**!

Esempio: Consideriamo `int calcolaSconto(int spesa)`

- **Test obligation** generate con metodo della frontiera
  - Provare tutti gli intorni delle cifre indicate
- **Casi di test** che soddisfano le test obligation
  - $\langle 48,0,\cdot\rangle, \langle 49,7,\cdot\rangle, \langle 50,8,\cdot\rangle$
  - $\langle 78,12,\cdot\rangle, \langle 79,14,\cdot\rangle, \langle 80,14,\cdot\rangle$
  - $\langle 98,18,\cdot\rangle, \langle 99,20,\cdot\rangle, \langle 100,20,\cdot\rangle$

| Spesa | Sconto totale |
|---|---|
| Da € 49 di acquisto | 15% |
| Da € 79 di acquisto | 18% |
| Da € 99 di acquisto | 20% |

# SELEZIONE VALORI: CASI NON VALIDI

Per ogni input, si definiscono anche i **casi non validi**

- devono generare un **errore**

Esempio: Consideriamo `int calcolaSconto(int spesa)`

- **Test obligation** generate con metodo dei casi non validi
  - Spesa negativa
- **Casi di test** che soddisfano le test obligation
  - $\langle -1, eccezione, \cdot \rangle, \langle -100, eccezione, \cdot \rangle, \ldots$

| Spesa | Sconto totale |
|---|---|
| Da € 49 di acquisto | 15% |
| Da € 79 di acquisto | 18% |
| Da € 99 di acquisto | 20% |

# SELEZIONE VALORI: RANDOM

Generare in modo **automatico** e **casuale** un insieme **grande a piacere** di valori

- Generazione a «**costo zero**»
- Applicabile se costa poco l'esecuzione
- **Non** sempre **ripetibile**
- Non considera i casi limite e può essere difficile trovare l'output atteso

Esempio: Trovare le soluzioni della seguente equazione

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

Quasi impossibile che il caso con $a = 0 \wedge b = 0$ sia generato in modo casuale

# SELEZIONE VALORI: CATALOGO

L'**esperienza** acquisita nel definire **casi di test** può essere collezionata in un **catalogo**

- Per rendere più veloce il processo e automatizzare alcune decisioni riducendo l'errore umano
- elencando **tutti i casi** che devono essere considerati per ciascun possibile **tipo di variabile**

Esempio: Consideriamo una funzione con un input in un dato 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 of the interval*
- 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

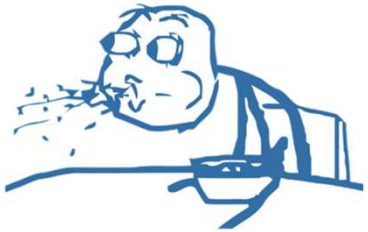
Al crescere del numero degli input, il prodotto cartesiano dei casi di test individuati può diventare ingestibile, per il fenomeno di **esplosione combinatoria**

Esempio: consideriamo `foo(a,b,c,d,e)`

- Dominio di `a` e `b` ripartibile in 8 classi (di cui una di valori non validi → errore)
- Dominio di `c` e `d` ripartibile in 4 classi (di cui una di valori non validi → errore)
- Dominio di `e` ripartibile in 7 classi (di cui una di valori non validi → errore)

Se prendiamo un rappresentante per classe e facciamo il prodotto cartesiano

$$8 * 8 * 4 * 4 * 7 = 7168 \text{ casi possibili}$$



Servono **strategie** per generare casi di test significativi in modo sistematico

- Tecniche per ridurre l'esplosione combinatoria, come **vincoli** e **pairwise testing**

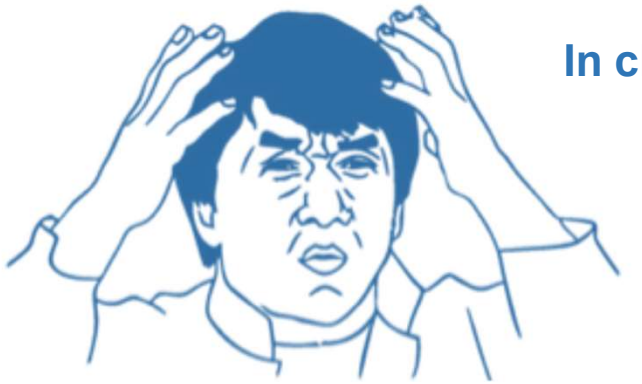
# TESTING COMBINATORIO: VINCOLI

Una possibilità è quella di introdurre **vincoli**

- per **limitare** il numero di test ottenuti generando tutte le **combinazioni** di valori possibili

Tre strategie a **vincoli** per ridurre il numero di possibili combinazioni

- vincoli di **errore** (o **error constraints**)
- vincoli di **proprietà**
- **singoletti**



**In che senso?**

# TESTING COMBINATORIO: VINCOLI (CONT.)

Vincoli di **errore** → **un solo caso** di errore (ovvero di **input non valido**) per ogni **posizione**

Esempio: consideriamo di nuovo `foo(a,b,c,d,e)`

- Viene preso un solo caso, per ogni posizione, con input non valido (5 casi)
- Una classe in meno da considerare per ogni dominio, ovvero

$5 + 7 * 7 * 3 * 3 * 6 = 2651$ casi possibili
(invece di $8 * 8 * 4 * 4 * 7 = 7168$)

# TESTING COMBINATORIO: VINCOLI (CONT.)

Vincoli di **proprietà** → definizione di **proprietà comuni** per **classi di equivalenza diverse** (usate poi per guidare la combinazione dei test)

Esempio: consideriamo di nuovo `foo(a,b,c,d,e)` e definiamo dei vincoli *property/if property* su

- Parametro `a`
  - classe 1, classe 2, classe 3, classe 4 [property: negativi]
  - classe 5, classe 6, classe 7 [property: positivi]
  - ~~classe8 [error]~~ // scartata, perché già considerata con error constraints
- Parametro `b`
  - classe 1, classe 3, classe 5, classe 7 [if: negativi]
  - classe 2, classe 4, classe 6 [if: positivi]
  - ~~classe8 [error]~~ // scartata, perché già considerata con error constraints

- Riduciamo a

$$5 + (4 * 4 + 3 * 3) * 3 * 3 * 6 = 5 + 1350 = 1355$$ casi possibili

(invece dei $7168$ originali)

# TESTING COMBINATORIO: VINCOLI (CONT.)

**Singoletti** → testare **un solo valore** per uno o più parametri

Esempio: consideriamo di nuovo `foo(a,b,c,d,e)`

- Applichiamo il metodo del singoletto al parametro `e`
- Riduciamo a

$$5 + (4 * 4 + 3 * 3) * 3 * 3 * 1 = 5 + 225 = 230$$ casi possibili

(invece dei $7168$ originali)

N.B. Rispetto agli error constraint, qui si fissa quel parametro, in error anche tutti gli altri

# TESTING COMBINATORIO: PAIRWISE TESTING

Introdurre vincoli è utile se i vincoli che imponiamo sono **reali vincoli del dominio** (e non se li aggiungiamo al solo scopo di limitare le combinazioni)

In alternativa, meglio optare per **pairwise testing** (ovvero basato su coppie di variabili)

- Generazione di tutte le combinazioni solo per i sottoinsiemi di 2 variabili
- Più in generale, per i sottoinsiemi di $k$ variabili, con $k < n$ (pairwise quando $k = 2$)

Esempio: consideriamo di nuovo `foo(a,b,c,d,e)` e applichiamo pairwise testing

- Generazione di combinazioni solo per le possibili coppie di parametri
- Tutte le combinazioni: $8 * 8 * 4 * 4 * 7 = 7168$ casi possibili
- Quanto si risparmia con il pairwise testing?

  $8 * 8 + 8 * 4 + 8 * 4 + 8 * 7 + 8 * 4 + 8 * 4 + 8 * 7 + 4 * 4 + 4 * 7 + 4 * 7 = 376$ casi possibili

(in realtà sono anche meno se generiamo le combinazioni in maniera efficiente)

# TESTING COMBINATORIO: PAIRWISE TESTING (CONT.)

Un **altro esempio**

| Display Mode | Language | Fonts |
|---|---|---|
| full-graphics | English | Minimal |
| text-only | French | Standard |
| limited-bandwidth | Spanish | Document-loaded |
| | Portuguese | |

| Color | Screen size |
|---|---|
| Monochrome | Hand-held |
| Color-map | Laptop |
| 16-bit | Full-size |
| True-color | |

Caso di test: text-only, laptop, standard

# TESTING COMBINATORIO: PAIRWISE TESTING (CONT.)

- Il prodotto cartesiano dei possibili valori di `display mode`, `screen size` e `fonts` genera

$3^3 = 27$ combinazioni possibili

- Il prodotto cartesiano ristretto ai valori di `display mode` e `screen size` genera

$3^2 = 9$ combinazioni possibili

- Poi dobbiamo generare anche tutte le combinazioni per le coppie che includono `fonts`
  - `fonts` e `screen size`
  - `fonts` e `display mode`

**Come farlo in modo efficiente?**

- Dopo aver generato le combinazioni per la prima coppia,
- il valore del terzo parametro può essere aggiunto in modo da coprire anche tutte le combinazioni delle coppie che includono `fonts`

# TESTING COMBINATORIO: PAIRWISE TESTING (CONT.)

| *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 |

**Attenzione**: la generazione efficiente di combinazioni che coprano tutte le coppie

- è **impossibile da fare a mano** per molti parametri con molti valori
- ma può essere fatta con **euristiche**


Q
A

# RIFERIMENTI

**Contenuti**

- **Capitolo 20** di "Software Engineering" (G. C. Kung, 2023)

**Approfondimenti**

- **Capitoli 9, 10 e 11** di "Software Testing and Analysis: Process, Principles and Techniques" (M. Pezzè, M. Young, 2008)