



UNIVERSITÀ DEGLI STUDI
DI MILANO

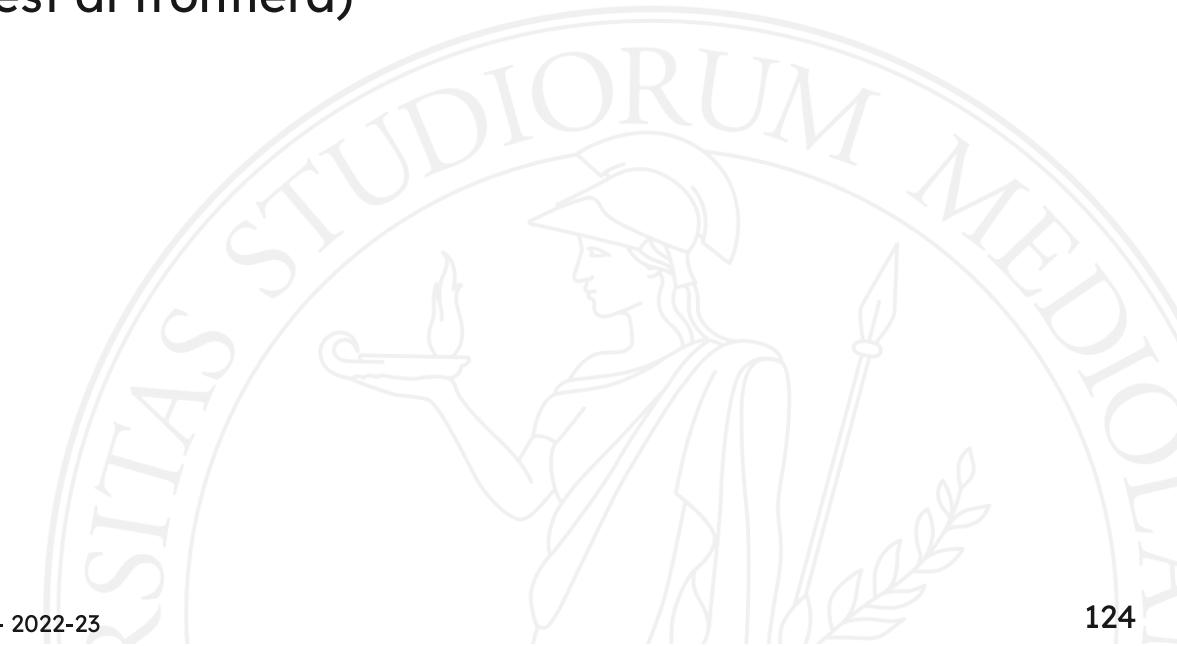
Verifica e Convalida

Test funzionale

- Non c'è (o meglio non si sfrutta) conoscenza del codice (*black box*)
- Può essere l'unico approccio possibile (test di validazione del lavoro di committente esterno)
- I dati di test possono essere derivati dalle specifiche (requisiti funzionali)
 - ci si concentra sul dominio delle informazioni invece che sulla struttura di controllo
- Permette di identificare errori non sintetizzabili con criteri strutturali
 - funzionalità non implementata
 - cammino di flusso dimenticato
 - si pone come obiettivo anche trovare errori di interfaccia e di prestazioni

Tecniche

- Metodi basati su grafi
- Suddivisioni del dominio in classi di equivalenza
 - Category partition
- Analisi dei valori limite (test di frontiera)
- Collaudo per confronto



Testing delle interfacce

- Tipi di interfacce:
 - a invocazione con parametri
 - a condivisione di memoria
 - a metodi sincroni
 - a passaggio di messaggi
- Tipi di errori
 - sbaglio nell'uso dell'interfaccia
 - Ordine o tipo dei parametri
 - assunzioni sbagliate circa ciò che la funzione si attende

Classi di equivalenza

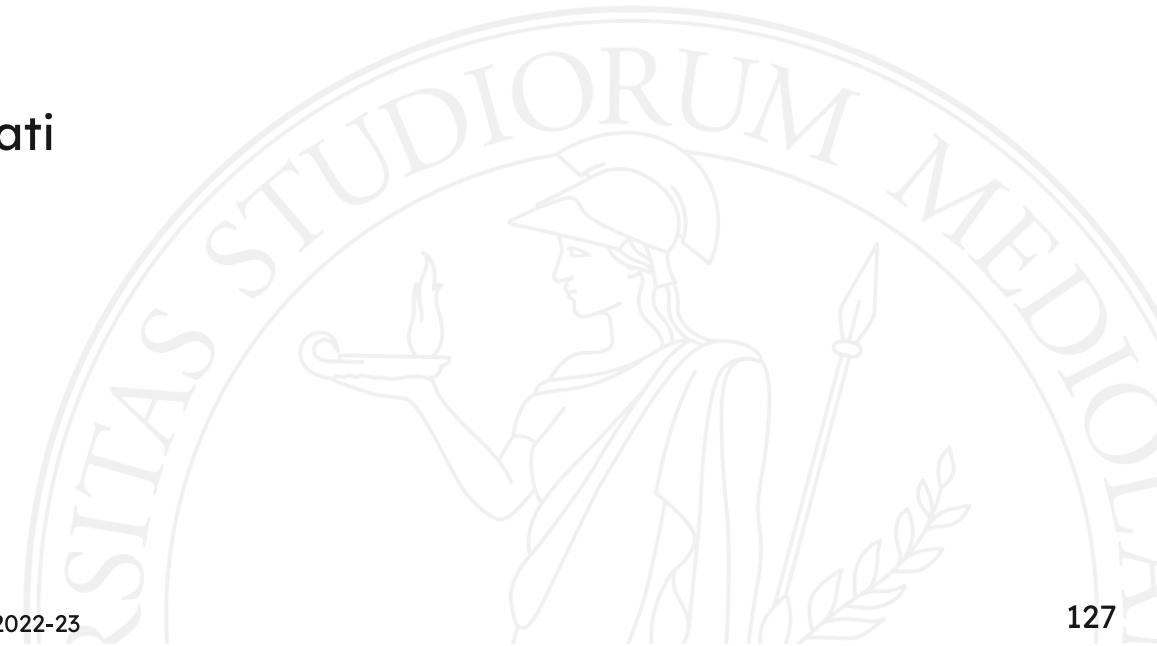
- Si basa sulla suddivisione del dominio dei dati in ingresso in classi di dati, dalle quali derivare casi di test
 - una classe di dati è un insieme i cui componenti *dovrebbero* essere trattati in maniera analoga dal programma
- Si cerca quindi di individuare casi di test che rivelino (eventuali) classi di errori
 - es. elaborazione scorretta per numeri negativi
 - o per numeri molto grandi

Classi di equivalenza

Una classe di equivalenza rappresenta un insieme di stati validi o non validi per i dati in input e un insieme di stati validi per i dati di output

Un dato può essere ad esempio:

- un valore
- un intervallo
- un insieme di valori correlati



Suddivisione in classi

- Se ci si aspetta un valore specifico vengono definite una classe valida ed una non valida. es. un **codice PIN**:

I PIN corretto

II numero di 4 cifre qualsiasi \neq PIN

- Se ci si aspetta un valore in un intervallo, vengono definite una classe di equivalenza valida e due non valide. es. **[100,700]**:

I numero tra 100 e 700

II numero minore di 100

III numero maggiore di 700

Test di frontiera

- Gli errori tendono ad accumularsi ai limiti del dominio
 - selezionare casi di test che esercitano i valori limite
- È complementare a classi di equivalenza:
 - non seleziono un elemento a caso della classe, ma uno ai confini



Category partition

È un particolare metodo di suddivisione in classi di equivalenza usabile a diversi livelli di granularità

- test di unità, di integrazione, di sistema

È composto dai seguenti passi:

1. Analizzare le specifiche

- Identificare le *unità funzionali* individuali che possono essere verificate singolarmente.
- Per ogni unità identificare le caratteristiche (*categorie*) dei parametri e dell'ambiente

2. scegliere dei valori (scelte) per le categorie

3. Determinare eventuali vincoli tra le scelte

4. (Scrivere test e documentazione)

Un esempio

Command

find

Syntax

find < pattern >< file >

Function

The find command is used to locate one or more instances of a given pattern in a file. All lines in the file that contain the pattern are written to standard output. A line containing the pattern is written only once, regardless of the number of times the pattern occur in it.

The pattern is any sequence of characters whose length does not exceed the maximum length of a line in the file. To include a blank in the pattern, the entire pattern must be enclosed in quotes (""). To include a quotation mark in the pattern, two quotes ("""") in a row must be used.



Passo 1: analizzare le specifiche

Find è una funzione individuale che può essere verificata separatamente

Parametri: pattern e file

Caratteristiche parametro pattern

- *esplicite* (immediatamente derivabili dalle specifiche)
 - lunghezza del pattern
 - pattern tra doppi apici
 - pattern contenente spazi
 - pattern contenente apici
- *implicite* (nascoste nelle specifiche)
 - pattern tra apici con/senza spazi
 - più apici successivi inclusi nel pattern

file (parametro o ambiente?)

- nome file è un *parametro*
- contenuto file è *ambiente*

Caratteristiche parametro nomefile

- implicite
 - caratteri nel nome ammissibili o meno
 - file esistente (con permessi di lettura)

Caratteristiche ambiente file

- esplicite
 - numero occorrenze pattern nel file
 - max numero occorrenze pattern in una linea
 - max lunghezza linea
- implicite
 - pattern sovrapposti
 - tipo del file

Passo 2

Determinare i singoli casi significativi per ogni categoria identificata

È importante l'esperienza (know-how)

- vanno identificati tutti e soli i casi significativi

Parametro pattern

Dimensione del pattern: vuoto, un solo carattere, più caratteri, più lungo di almeno una linea del file

Presenza di apici: pattern tra apici, pattern non tra apici, pattern tra apici errati

Presenza di spazi: nessuno, uno, molti

Presenza apici interni: nessuno, uno, molti

Passo 3

- Una combinazione di scelte per ogni parametro/ambiente rappresenta un caso di test plausibile (partizione)
- Generare un test per ogni combinazione delle classi fino qui identificate porta a 1.944 casi di test (troppi!)
- Come ridurli?



Approccio combinatorio

```
public class Main {  
    public static void main(String[] args) {  
        AllPairs allPairs =  
            new AllPairs.AllPairsBuilder()  
                .withParameters(Arrays.asList(  
                    new Parameter("Category",  
                        "Sell", "Buy"),  
                    new Parameter("Location",  
                        "Mumbai", "Dehli"),  
                    new Parameter("Brand",  
                        "Mercedes", "BMW", "Audi"),  
                    new Parameter("Registration",  
                        "Valid", "Not Valid"),  
                    new Parameter("Order Type",  
                        "Ebooking", "InStore"),  
                    new Parameter("Order Time",  
                        "Working", "NotWorking")))  
                .printEachCaseDuringGeneration()  
                .build();  
    }  
}
```

- <https://www.softwaretestinghelp.com/what-is-pairwise-testing/>
- <https://www.pairwise.org/tools.html>
- <https://pairwise.teremokgames.com/2yarw/>
- Category=Sell, Location=Mumbai, Brand=Mercedes, Registration=Valid, Order Type=Ebooking, Order Time=Working
- Category=Buy, Location=Dehli, Brand=BMW, Registration=Not Valid, Order Type=InStore, Order Time=Working
- Category=Buy, Location=Mumbai, Brand=Audi, Registration=Not Valid, Order Type=Ebooking, Order Time=NotWorking
- Category=Sell, Location=Dehli, Brand=Audi, Registration=Valid, Order Type=InStore, Order Time=Working
- Category=Sell, Location=Dehli, Brand=BMW, Registration=Valid, Order Type=Ebooking, Order Time=NotWorking
- Category=Buy, Location=Mumbai, Brand=Mercedes, Registration=Not Valid, Order Type=InStore, Order Time=Working
- Category=Buy, Location=Mumbai, Brand=BMW, Registration=Valid, Order Type=InStore, Order Time=Working
- Category=Sell, Location=Dehli, Brand=Mercedes, Registration=Not Valid, Order Type=InStore, Order Time=Working
- Category=Sell, Location=Dehli, Brand=Mercedes, Registration=Not Valid, Order Type=Ebooking, Order Time=NotWorking

determinare vincoli

Più combinazioni di alcune scelte particolari possono essere poco significative:

- determinare vincoli per eliminare combinazioni meno significative
 - *proprietà* (ad esempio *NotEmpty* o *Quoted*)
 - *se* (limita l'uso di un valore solo ai casi in cui è definita una proprietà)
 - *single*
 - file che non contiene occorrenze del pattern cercato (che produce sempre lo stesso risultato indipendentemente dal tipo di pattern cercato)
 - *error*
 - nome del file omesso (che corrisponde ad un errore)

Passo 4

È possibile stimare il numero di casi di test e generare specifiche di test in modo automatico a partire da scelte e vincoli

Numero e distribuzione di vincoli possono essere determinati iterativamente stimando i casi di test

- generando le specifiche a partire da scelte e vincoli quando la stima dei casi di test è ragionevole
- valutando i test generati ed introducendo nuovi vincoli se in presenza di forti ridondanze

Testing funzionale e OO ?

Visto che non si basa sul codice ...

- Problemi nella fase di integrazione
 - in generale non esiste la struttura gerarchica che possa guidare l'integrazione delle unità
- È possibile però trovare dei cluster significativi
 - Use cases e scenari
 - Sequence Diagram e copertura dei thread de messaggi
 - State Diagram

Software inspection

- Tecnica manuale per individuare e correggere gli errori basata su di una attività di gruppo
- Fagan Code Inspections
 - la più diffusa tra le tecniche di ispezione (più rigorosa e definita)
 - estesa a fase di progetto e raccolta requisiti

Ruoli

Moderatore:

- in genere preso a prestito da un altro processo, coordina i meeting, sceglie i partecipanti, controlla il processo **Readers, Testers:**
- leggono il codice al gruppo, cercano difetti **Autore:**
- partecipante passivo; risponde ad eventuali domande

Software Inspection Process

- Planning
 - Il moderatore sceglie i partecipanti e fissa gli incontri
- Overview
 - per fornire il background e assegnare i ruoli
- Preparation
 - Attività svolte offline per la comprensione del codice o della struttura del sistema
- Inspection
- Rework
 - l'autore si occupa dei difetti individuati
- Follow-up:
 - possibile re-ispezione

Ispezione

Goal: trovare e registrare il maggior numero di difetti, ma non correggerli

- al massimo 2 sessioni di 2 ore al giorno approx. 150 source lines all'ora

Approccio: parafrasare linea per linea

- risalire allo scopo del codice a partire dal sorgente
- possibile anche “test a mano” trovare e registrare difetti, ma non correggerli
- checklist

Checklist - esempio NASA

- Circa 2.5 pagine per il C, 4 per FORTRAN
 - Divise in: Functionality, Data Usage, Control, Linkage, Computation, Maintenance, Clarity
- Esempio:
 - Does each module have a single function?
 - Does the code match the Detailed Design?
 - Are all constant names upper case?
 - Are pointers not typecast (except assignment of NULL)?
 - Are nested "#include" files avoided?
 - Are non-standard usage isolated in subroutines and well documented?
 - Are there sufficient comments to understand the code?

Incentive structure

- Difetti trovati non devono essere utilizzati per la valutazione del personale
 - Il programmatore non va incentivato a nascondere difetti
- Difetti trovati dal test (dopo l'ispezione) sono usati per la valutazione del personale
 - Il programmatore è incentivato a trovare tutti i difetti durante l'ispezione

Variante: Active Design Reviews

- Osservazione:
 - Un revisore non preparato può stare seduto tranquillamente e non dire niente
- Variante al processo:
 - Scegliere revisori con adeguata esperienza
 - Diversi revisori per diversi aspetti
 - L'autore fa domande al revisore (checklist)
 - Il revisore dovendo rispondere è costretto a partecipare

Automazione dell'ispezione

Sebbene sia una tecnica manuale esistono tool di supporto:

- Controlli banali (e.g., formattazione)
- Riferimenti: Checklists, Standard con esempi
- Aiuti alla comprensione del codice
 - Tool comuni a quelli di attività di reengineering
 - Evidenziazione di parti rilevanti
 - Navigazione nel codice
 - Diversi tipi di rappresentazione dei dati e delle architetture
 - Annotazioni & comunicazioni
 - Guida al processo e rinforzo

Funziona ?

La pratica ci dice che è *cost-effective* Perché?

- Processo rigoroso e dettagliato
- Basato su accumulo di esperienza (es. Checklist) si auto migliora
- Aspetti sociali del processo (riguardo all'autore soprattutto)
- Si considera l'intero dominio dei dati
- È applicabile anche a programmi incompleti

Limiti

- Livello del test: solo a livello di unità
- Non incrementale: evoluzione del software?

Fagan's Law (L17)

Inspections significantly increase productivity, quality, and project stability



Confronto tra le varie tecniche

Authors	Average Percentage of defects found by subjects applying		
	Inspection (Code Reading)	Functional Testing	Structural Testing
Hetzell [12]	37.3	47.7	46.7
Myers [21]	30.0	36.0	38.0
Basili & Selby [3]	54.1	54.6	41.2
Kamsties & Lott [17] (1. Replication)	40.0	37.0	36.0
Kamsties & Lott [17] (2. Replication)	50.3	53.4	33.5
Wood et al. [28]	43.41	55.05	57.87

... e metterle insieme ?

Table 2. Defect-removal efficiency under 16 permutations of four factors.

	Factors used				Results		
	Formal design inspection	Formal code inspection	Formal quality assurance	Formal testing	Worst	Median	Best
1		(None used)			30%	40%	50%
2			✓		32%	45%	55%
3				✓	37%	53%	60%
4		✓			43%	57%	66%
5	✓				45%	60%	68%
6			✓	✓	50%	65%	75%
7		✓	✓		53%	68%	78%
8		✓		✓	55%	70%	80%
9	✓		✓		60%	75%	85%
10	✓			✓	65%	80%	87%
11	✓	✓			70%	85%	90%
12		✓	✓	✓	75%	87%	93%
13	✓		✓	✓	77%	90%	95%
14	✓	✓	✓		83%	95%	97%
15	✓	✓	✓	✓	85%	97%	99%
16	✓	✓	✓	✓	95%	99%	99.99%

Hetzell-Myers's Law (L20)

A combination of different V&V methods outperforms any single method alone

Vantaggi gruppo di test autonomo

Weinberg's Law (L23)
A developer is unsuited to test his or her code

- Aspetti tecnici
 - maggiore specializzazione
 - conoscenza delle tecniche e degli strumenti
- Aspetti psicologici
 - distacco dal codice
 - test indipendente da conoscenza codice
 - attenzione ad aspetti dimenticati
 - indipendenza della valutazione



Svantaggi gruppo di test autonomo

- Aspetti tecnici
 - progressiva perdita di capacità di progetto e codifica
 - minore conoscenza dei requisiti
- Aspetti psicologici
 - possibile pressione negativa su team sviluppo
 - possibile gestione scorretta delle responsabilità

Possibili alternative

- Rotazione del personale
 - permette di evitare progressivo depauperamento tecnico dovuto a eccessiva specializzazione
 - permette di evitare svuotamento dei ruoli
 - aumenta i costi di formazione
 - aumenta le difficoltà di pianificazione
- Condivisione del personale
 - permette di supplire a scarsa conoscenza del prodotto in esame
 - aumenta le difficoltà di gestione dei ruoli

Modelli statistici

- relazione statistica tra metriche e
 - presenza errori (per classi di errori)
 - numero errori (per classi di errori)

Possibile predire distribuzione errori per modulo

Pareto-Zipf-type Laws (L24)

Approximately 80 percent of defects come from 20 percent of modules

Debugging

- Mira a localizzare e rimuovere le anomalie che sono le cause di malfunzionamenti riscontrati nel programma
- Non deve essere usato per rilevare malfunzionamenti
- L'attività è definita per un programma e un insieme di dati che causano malfunzionamenti nel programma
 - Si basa su riproducibilità del malfunzionamento
 - Va verificato che il “malfunzionamento” non sia dovuto a specifiche errate

“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.” — Brian W. Kernighan

Problemi

- Non *sempre* facile stabilire una relazione anomalia-malfunzionamento
- Non esiste una relazione biunivoca tra anomalie e malfunzionamento

*10 little Indian boys went out to dine;
One choked his little self and then there were 9.*

*9 little Indian boys sat up very late;
One overslept himself and then there were 8.*

*10 little bugs were in the code.
Take one down, patch it around.*

27 little bugs were in the code.

Tecnica Naïve

Consiste nell'introdurre nel modulo in esame comandi di uscita che stampino il valore intermedio assunto dalle variabili

- facile da applicare (bastano un compilatore e un esecutore)
- richiede la modifica del codice (e quindi la sua rimodifica una volta individuata la anomalia)
- poco flessibile (modifica e compilazione per ogni nuovo stato)

Tecnica Naïve “avanzata”

Un miglioramento parziale si può ottenere sfruttando funzionalità del linguaggio

- ad esempio in C #ifdef e -D
- librerie di logging (con tipologia messaggi)
- asserzioni
 - possono essere viste anche come oracoli interni al codice

Dump di memoria

Consiste nel produrre una immagine esatta della memoria dopo passo di esecuzione

- non richiede modifica del codice
- spesso difficile per la differenza tra la rappresentazione astratta dello stato (legata alle strutture dati del linguaggio utilizzato) e la rappresentazione fornita dallo strumento
- viene prodotta una mole di dati per la maggior parte inutile

Debugging simbolico

Gli stadi intermedi sono prodotti usando una rappresentazione compatibile con quella del linguaggio usato

- Gli stati sono rappresentati come strutture dati e valori ad esse associati
- I debugger simbolici forniscono ulteriori strumenti (watch o spy monitor) che permettono di visualizzare il comportamento del programma in maniera selettiva
 - inserimento breakpoint, watch su variabili

Debugging per prova

Non solo la visualizzazione ma anche l'esame automatico degli stati ottenuti

- strumenti per verificare la “correttezza” degli stadi intermedi
- watch condizionali
- asserzioni a livello di monitor (invece che nel codice)
 - ad esempio si chiede al monitor di controllare che gli indici di un array siano sempre interni all'intervallo



Altre funzionalità debugger

- Gestire la granularità del passo di esecuzione:
 - singolo passo
 - entrare dentro a funzione
 - drop/reset del frame
- Modificare il contenuto di una variabile (o zona di memoria)
- Modificare codice
 - non sempre possibile... necessita ricompilazione ma poi si prosegue dal punto in cui ci si era interrotti
- Rappresentazioni grafiche dei dati

È possibile automatizzare il debugging?

<http://www.st.cs.uni-saarland.de/dd/>

<https://www.debuggingbook.org/html/DeltaDebugger.html>

Delta (differential) debugging

- Andreas Zeller (Saarland University, Saarbrucken, Germany)