# INTRODUZIONE ALLA FASE E AI CONCETTI DI VERIFICA E VALIDAZIONE

Prof. Mariacarla Staffa

# LASCIATE OGNI SPERANZA O VOI CHE ENTRATE

- Il problema della verifica di correttezza è difficile: è indecidibile, i.e. non esiste un algoritmo che lo risolva

# UN RISULTATO FONDAMENTALE

- Nel 1937 Alan Turing ha dimostrato che alcuni problemi non possono essere risolti da un algoritmo (programma)

- Tali problemi sono quelli che coinvolgono il problema della terminazione

## PROBLEMA DELLA TERMINAZIONE:

- esiste un algoritmo/programma che, presi in ingresso un qualsiasi altro programma e un input, stabilisca se il programma su tale input termina o no ?

# PROBLEMA DELLA TERMINAZIONE

Si assuma che esista un programma C che prende in input un programma (a) e un input per a che chiamiamo d

```
// halts() restituisce true se a(d) termina, false altrimenti
boolean C(a, d) {return halts(a(d));}
```

Dato che un programma è sua volta una sequenza di caratteri, si può invocare C(a,a). Si può quindi definire K(a) come segue

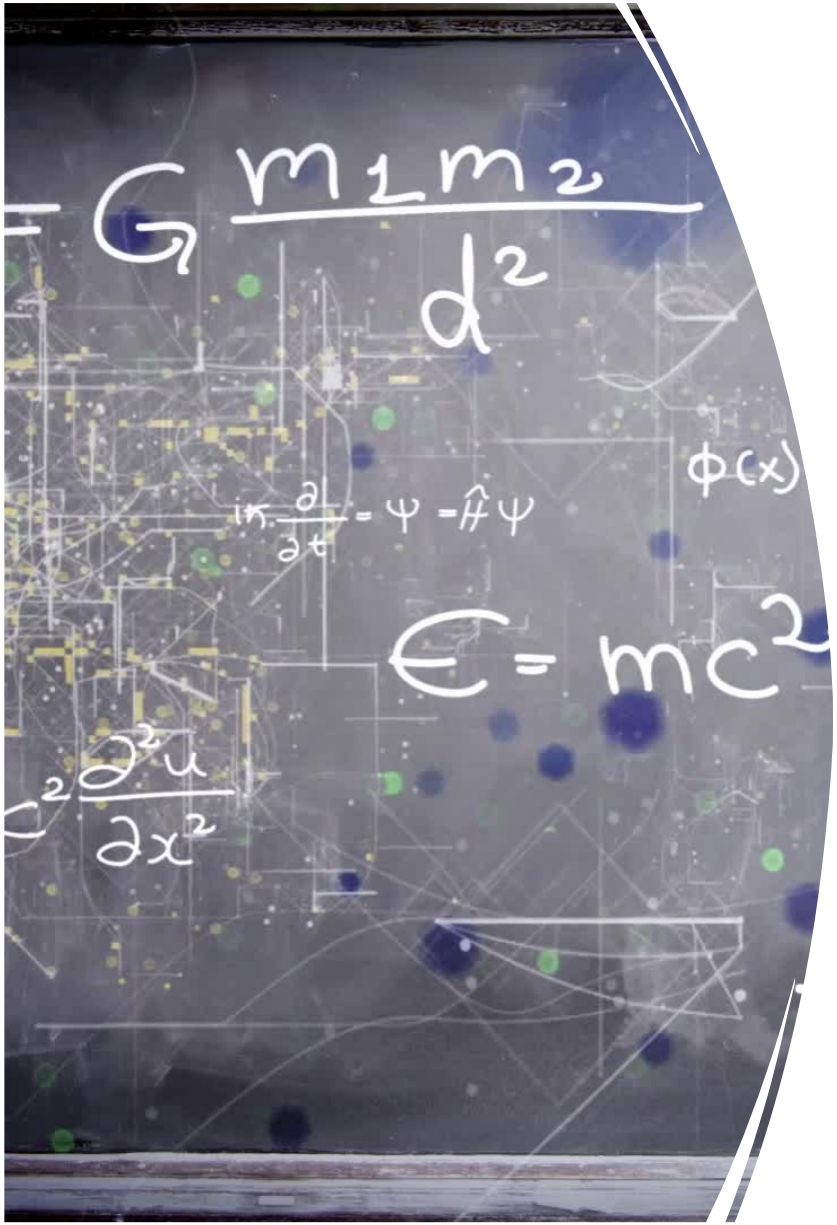
```
boolean K(a){
    if C(a,a) while(true){skip;};
    else return false;
    }
```

## Il programma K con input K termina?

**Il programma K con input K termina**, (restituendo il valore false) se e solo se C(K,K) è falso, ma C(K,K) è falso solo se halts(K(K)) è falso, **vale a dire se il programma K con input K non termina.**

K(K) termina se e solo se K(K) non termina. Contraddizione

- Non può esistere il programma C!




# UN RISULTATO FONDAMENTALE CONT'D

- In particolare, dobbiamo concludere che non esiste un programma P che per ogni programma Q e input D, dice se il programma Q sull'input D termina o no (Halting Problem).

```
while x>0
        do x=x+1;
y=27;
```

- Purtroppo non è solo un risultato teorico: quasi tutte le proprietà interessanti dei programmi incorporano l'halting problem

# INDECIDIBILITÀ

Esistono programmi che è possibile dimostrare corretti in tempo finito

```
public void printHW(){
            System.out.println("Hello, World");
}
```

Ne esistono altri per cui ciò non è possibile

Quindi, non esiste alcun programma P che prende in input altri programmi e PER OGNUNO di questi decide in tempo finito se è corretto o meno

# E' INDECIDIBILE, PER ESEMPIO

Dire con un algoritmo generale se un generico programma C vada in ciclo infinito su un generico input

```
public void printC(myHome Home) {
       Calzini calzini = myhome.calzini;
       while (not appaiati(calzini))
              calzini=appaia(calzini);
       System.out.println("Hello, World");
   }
```

# E' ANCHE INDECIDIBILE, PER ESEMPIO

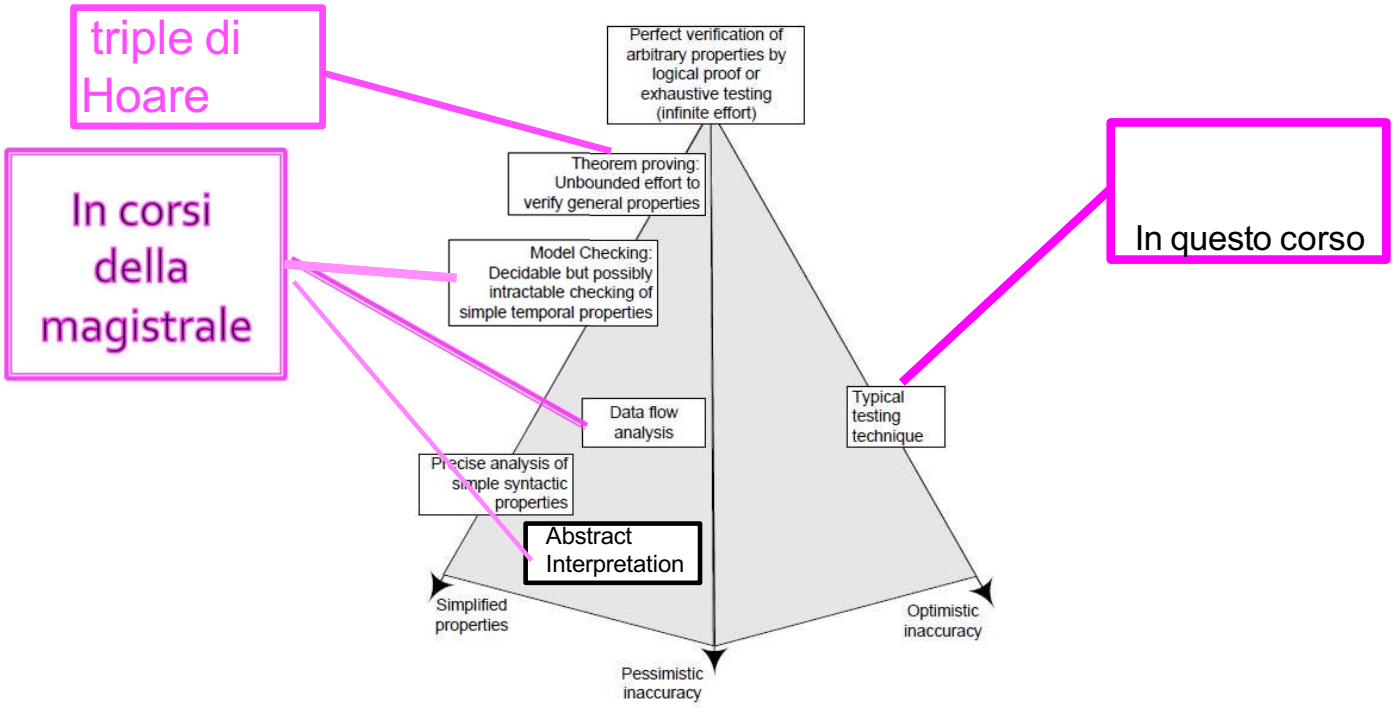
Dire se due programmi C producono lo stesso risultato in corrispondenza degli stessi dati di ingresso

- Per esempio i due metodi visti

```
public void printHW(int x){ System.out.println("Hello, World");
  }

public void printC(int x) { while (x>0) x=x+1;
      System.out.println("Hello, World");
  }
```

# VEDIAMO COSA POSSIAMO FARE



# TRIPLE DI HOARE

## Dove si nasconde il problema?

- La logica al primo ordine è indecidibile
  - In altre parole: Esiste un algoritmo che, per ogni formula F in logica al primo ordine, mi permetta di decidere in tempo finito se F è valida o meno?

    se cioè F oppure ¬ F?
- No, non esiste. Si possono enumerare (scrivere una dopo l'altra) tutte le formule valide, ma in tempo finito posso non arrivare a scrivere né F né ¬ F

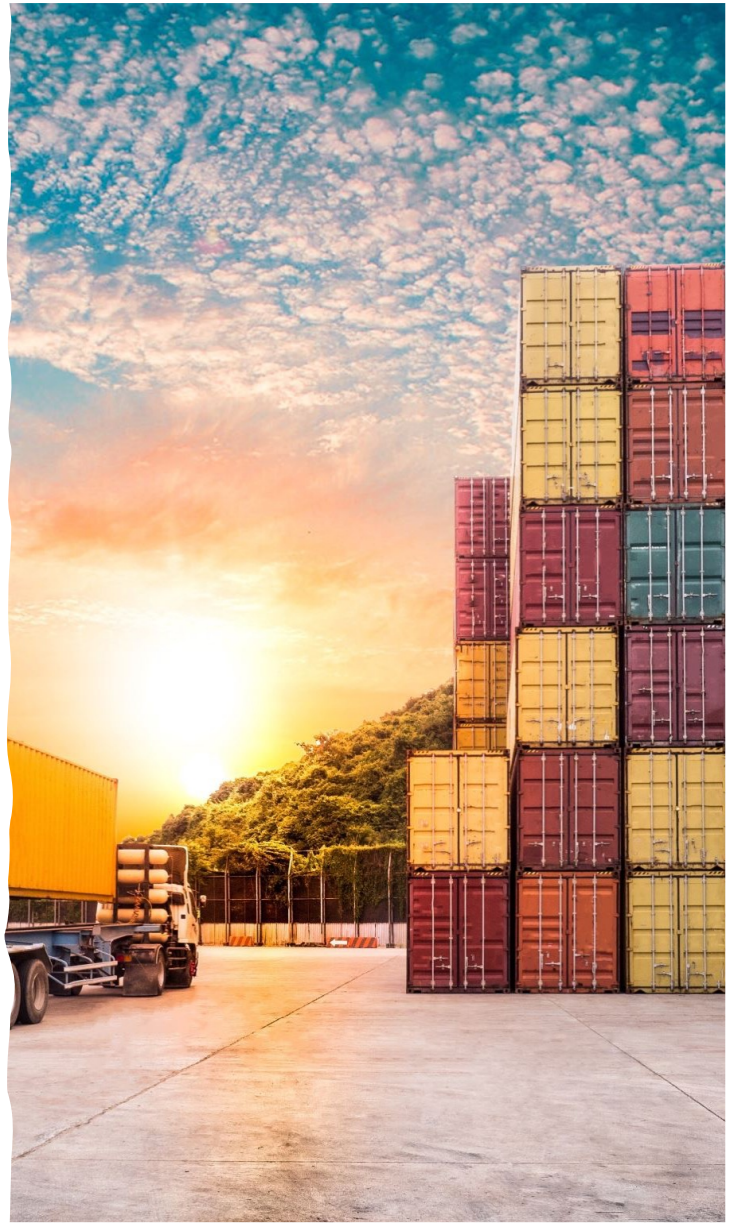
# ROADMAP

Concetti e terminologia

- Visualizzare il "quadro generale" della qualità del software nel contesto di un progetto di sviluppo software e organizzazione:
- attività di verifica e di validazione (V&V: verification and validation) del software
- la selezione e la combinazione di attività di V&V all'interno di un processo di sviluppo software.

# IL SW HA ALCUNE CARATTERISTICHE CHE RENDONO V&V PARTICOLARMENTE DIFFICILE

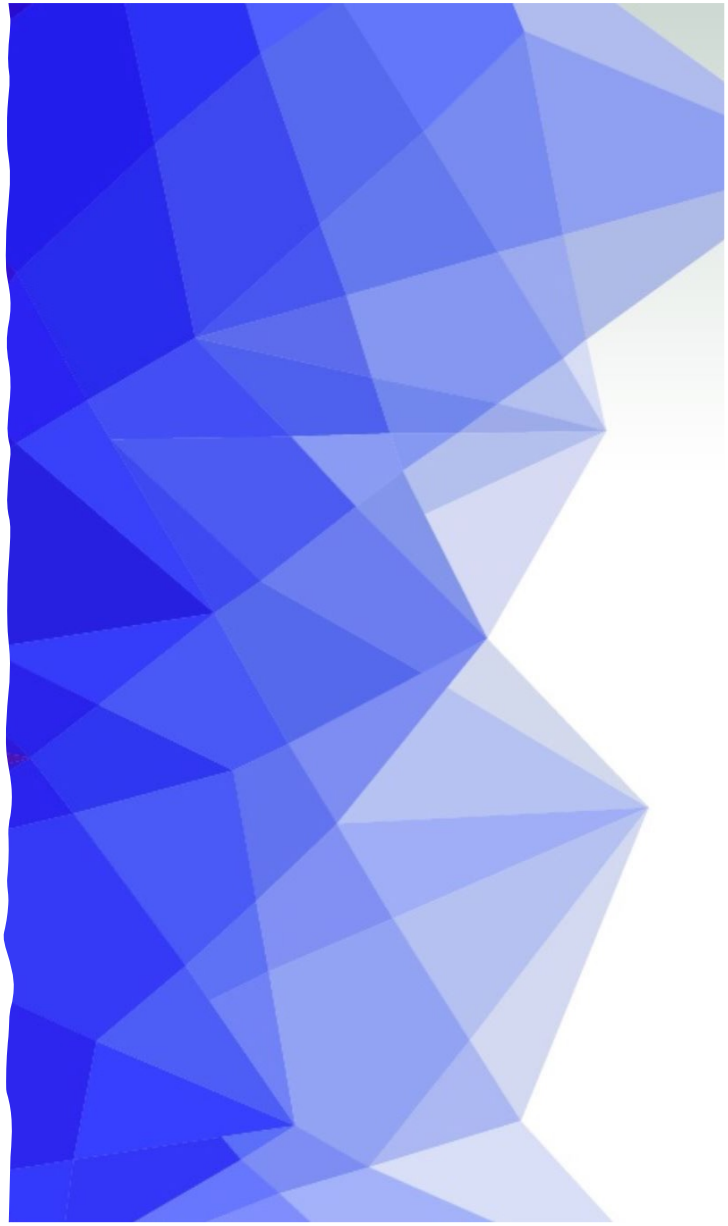
- requisiti di qualità diversi
- il sw è sempre in evoluzione distribuzione irregolare dei guasti
- non linearità, esempio:
  - Se un ascensore può trasportare un carico di 1000 kg, può anche trasportare qualsiasi carico minore:
  - se una procedura ordina correttamente un set di 256 elementi, potrebbe non riuscire su un set di 255 o 53 o 12 elementi, nonché su 257 o 1023.



# DIPENDENZA DAI LINGUAGGI

Nuovi approcci di sviluppo possono introdurre nuovi tipi di errori

- deadlock o race conditions per il software distribuito
- problemi dovuti al polimorfismo o al binding dinamico nel software object-oriented



# PROGETTARE LA FASE DI VERIFICA

- I progettisti della fase di verifica devono:
  - scegliere e programmare la giusta combinazione di tecniche
    - per raggiungere il livello richiesto di qualità
    - entro i limiti di costo
  - progettare una soluzione specifica che si adatta
    - al problema
    - ai requisiti
    - all'ambiente di sviluppo
- Senza poter contare su "ricette" fisse

# 5 DOMANDE DA USARE COME GUIDA

- Quando iniziare verifica e convalida? Quando sono complete?
- Quali tecniche applicare?
- Come possiamo valutare se un prodotto è pronto per essere rilasciato?
- Come possiamo controllare la qualità delle release successive?
- Come può essere migliorato il processo di sviluppo?

# 1. QUANDO INIZIARE VERIFICA E CONVALIDA? QUANDO SONO COMPLETE?

- Il testing non è una fase finale dello sviluppo software
    - L'esecuzione dei test è solo una piccola parte del processo di verifica e convalida
- V & V iniziano non appena decidiamo di creare un prodotto software
- V & V durano molto oltre la consegna dei prodotti
    - per tutto il tempo in cui il software è in uso
    - per far fronte alle evoluzioni e agli adattamenti alle nuove condizioni

# QUANDO INIZIARE VERIFICA E CONVALIDA?

- Dallo studio di fattibilità
  - Lo studio di fattibilità di un nuovo progetto deve tener conto delle qualità richieste e dell'impatto sul costo complessivo

- In questa fase, le attività correlate alla qualità comprendono:
  - analisi del rischio
  - definizione delle misure necessarie per valutare e controllare la qualità in ogni stadio di sviluppo
  - valutazione dell'impatto di nuove funzionalità e nuovi requisiti di qualità
  - valutazione economica delle attività di controllo della qualità: costi e tempi di sviluppo

# DOPO IL RILASCIO

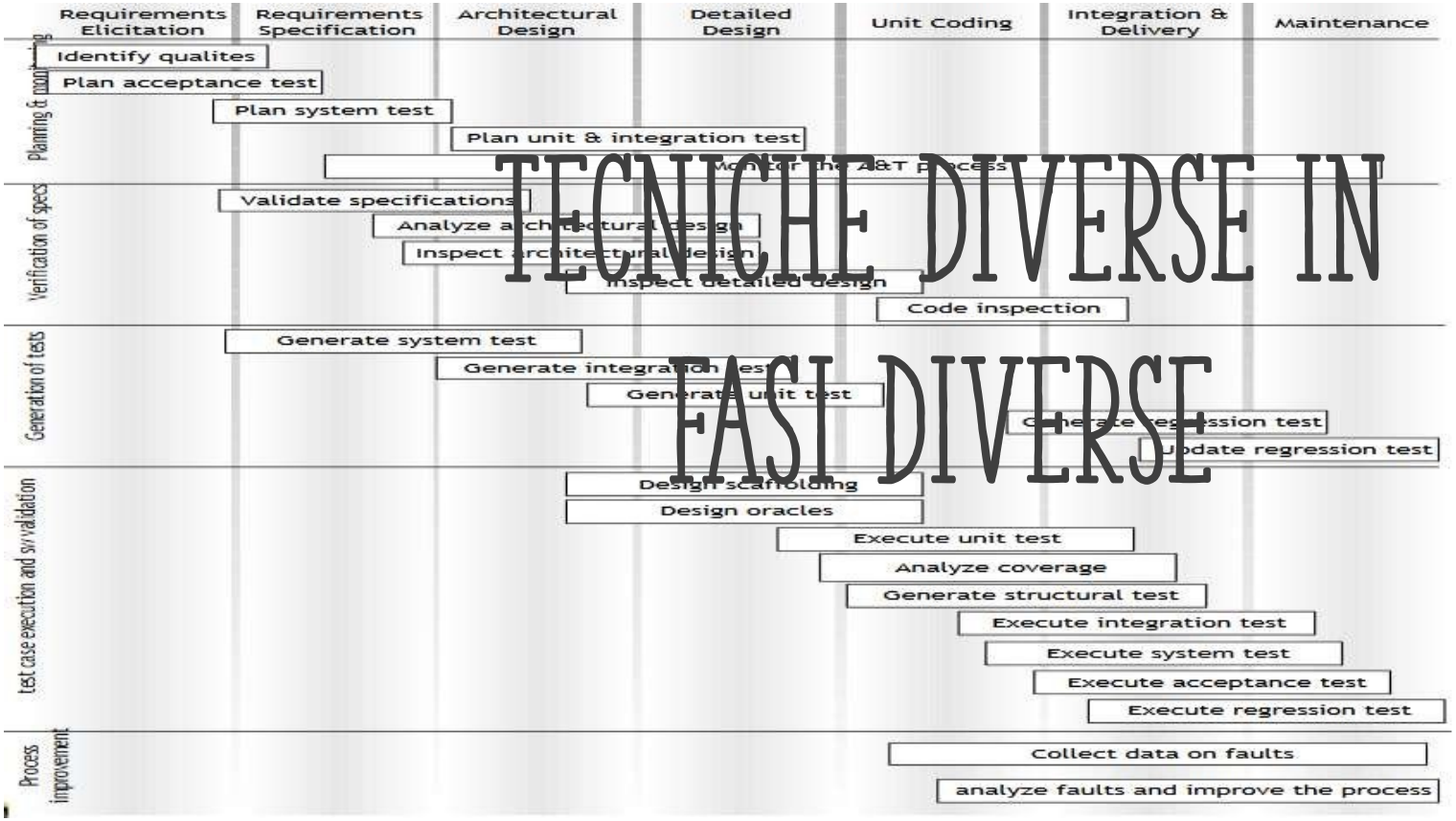
- Le attività di manutenzione comprendono:
  - analisi delle modifiche ed estensioni,
  - generazione di nuove suite di test per le funzionalità aggiuntive,
  - riesecuzione dei test per verificare la non regressione delle funzionalità del software dopo le modifiche e le estensioni
  - rilevamento e analisi dei guasti

# 2. QUALI TECNICHE APPLICARE?

Nessuna singola tecnica di analisi e testing (A & T) è sufficiente per tutti gli scopi.

Le principali ragioni per combinare diverse tecniche sono:

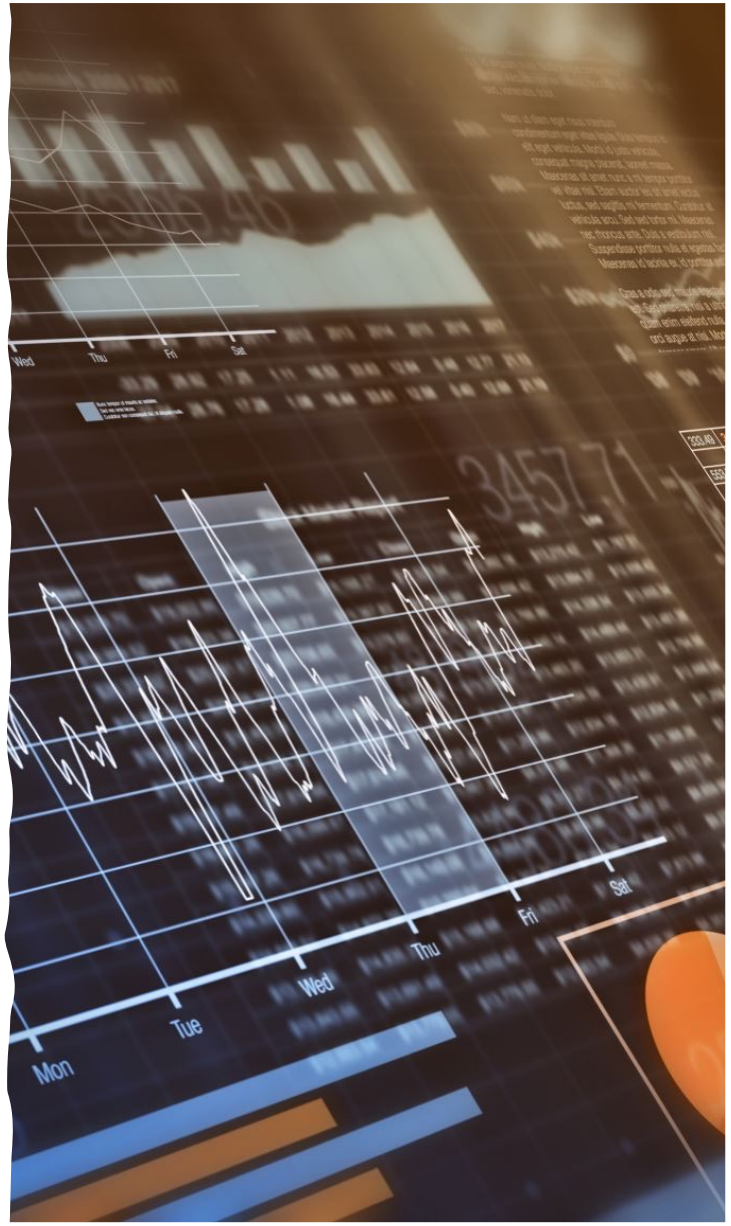
1. Efficacia per diverse classi di difetti: analisi statica invece di test per le race conditions
2. Applicabilità in diverse fasi del processo di sviluppo, per esempio: ispezione per la convalida dei requisiti iniziali
3. Differenzenegli scopi. Esempio: test statistico per misurare l'affidabilità
4. Compromessi in termini di costo e affidabilità: usare tecniche costose solo per requisiti di sicurezza




# TECNICHE DIVERSE IN FASI DIVERSE

# 3. COME POSSIAMO VALUTARE SE UN PRODOTTO È PRONTO PER ESSERE RILASCIATO?

- Alcune misure di dependability
  - La **disponibilità** misura la qualità di un sistema in termini di tempo di esecuzione rispetto al tempo in cui il sistema è giù
  - Il **tempo medio tra i guasti** (MTBF) misura la qualità di un sistema in termini di tempo tra un guasto e il successivo
  - **L'affidabilità** indica la percentuale di operazioni che terminano con successo




# 3. COME POSSIAMO VALUTARE SE UN PRODOTTO È PRONTO PER ESSERE RILASCIATO?

## Definire bene le misure

- Esempio: Applicazione e-shop realizzata con 100 operazioni
  - Il software funziona correttamente fino al punto in cui viene indicata una carta di credito: nel 50% dei casi viene addebitato l'importo sbagliato.
- Qual è l'affidabilità del sistema?
  - Se contiamo la percentuale di operazioni corrette , solo una operazione su 100 fallisce: il sistema è affidabile al 99%
  - Se contiamo le sessioni, solo il 50% affidabile

# 3. COME POSSIAMO VALUTARE SE UN PRODOTTO È PRONTO PER ESSERE RILASCIATO? ALFA E BETA TEST

## Alfa test:

- test eseguiti dagli sviluppatori o dagli utenti in ambiente controllato, osservati dall'organizzazione dello sviluppo

## Beta test:

- test eseguiti da utenti reali nel loro ambiente, eseguendo attività reali senza interferenze o monitoraggio ravvicinato

# 4. COME POSSIAMO CONTROLLARE LA QUALITÀ DELLE RELEASE SUCCESSIVE?

Attività dopo la consegna

- test e analisi del codice nuovo e modificato
- riesecuzione dei test di sistema
- memorizzazione di tutti i bug trovati
- test di regressione
    - Quasi automatico
- distinzione tra "major" e "minor" revisions
    - 2.0 vs 1.4
    - 1.5 vs 1.4

# TEST DI REGRESSIONE...

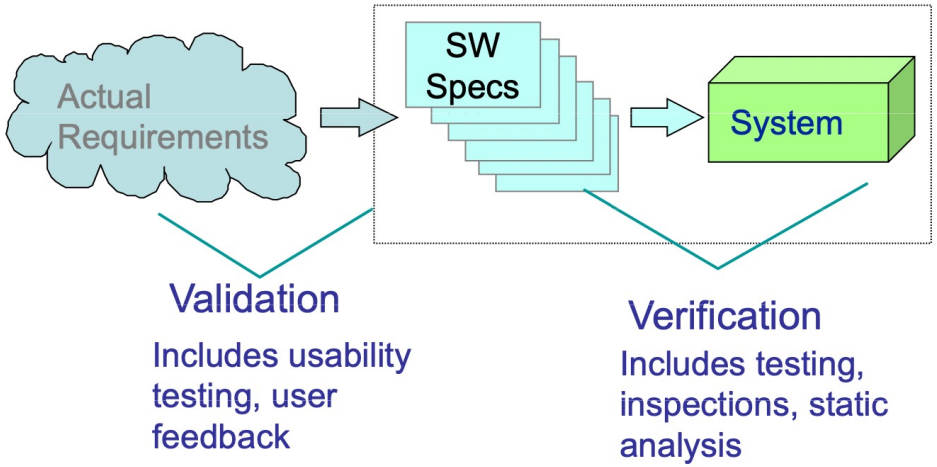
- Il **test di regressione**, o regression test, è una tipologia di software testing per verificare il corretto funzionamento del software dopo l'**implementazione di una nuova funzionalità** o la **risoluzione di eventuali bug**. In questi casi, la dipendenza tra funzionalità aggiunte ed esistenti può essere elevata e causare comportamenti**inattesi**. Per garantire la qualità del software, quindi, è essenziale verificare che il codice aggiunto non interferisca con le funzionalità esistenti.
- I test di regressione puntano a risolvere **un problema comune che gli sviluppatori devono affrontare** su base quotidiana: la comparsa di *regression bug* quando vengono introdotte modifiche. Nel caso in cui in un progetto non siano presenti rigidi **sistemi di controllo della versione**, diventa difficile rintracciare quale modifica abbia introdotto i problemi riscontrati, e quindi come risolverli.

# 5. COME PUÒ ESSERE MIGLIORATO IL PROCESSO DI SVILUPPO?

Si incontrano gli stessi difetti progetto dopo progetto

- identificare e rimuovere i punti deboli nel processo di sviluppo
  - Per esempio cattive pratiche di programmazione

- identificare e rimuovere i punti deboli del test e dell'analisi che consentono loro di non essere individuati



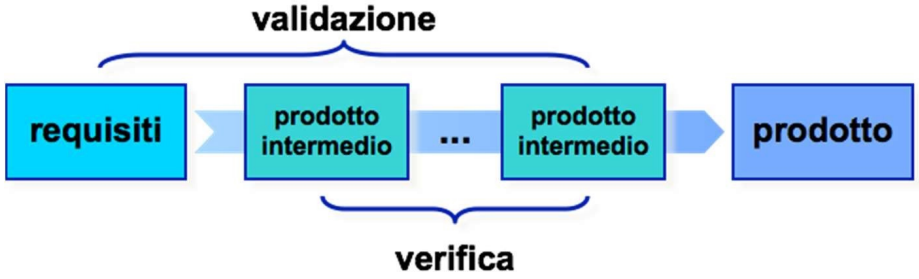

# VERIFICA VS CONVALIDA

- Validation: does the software system meets the user's real needs?

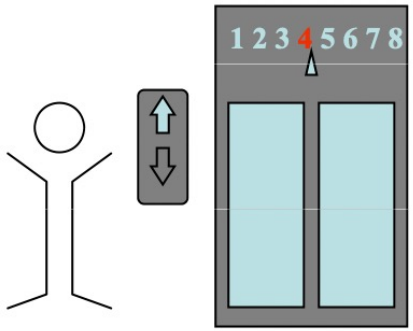
  are we building the right software?

- Verification: does the software system meets the requirements specifications?

  are we building the software right?



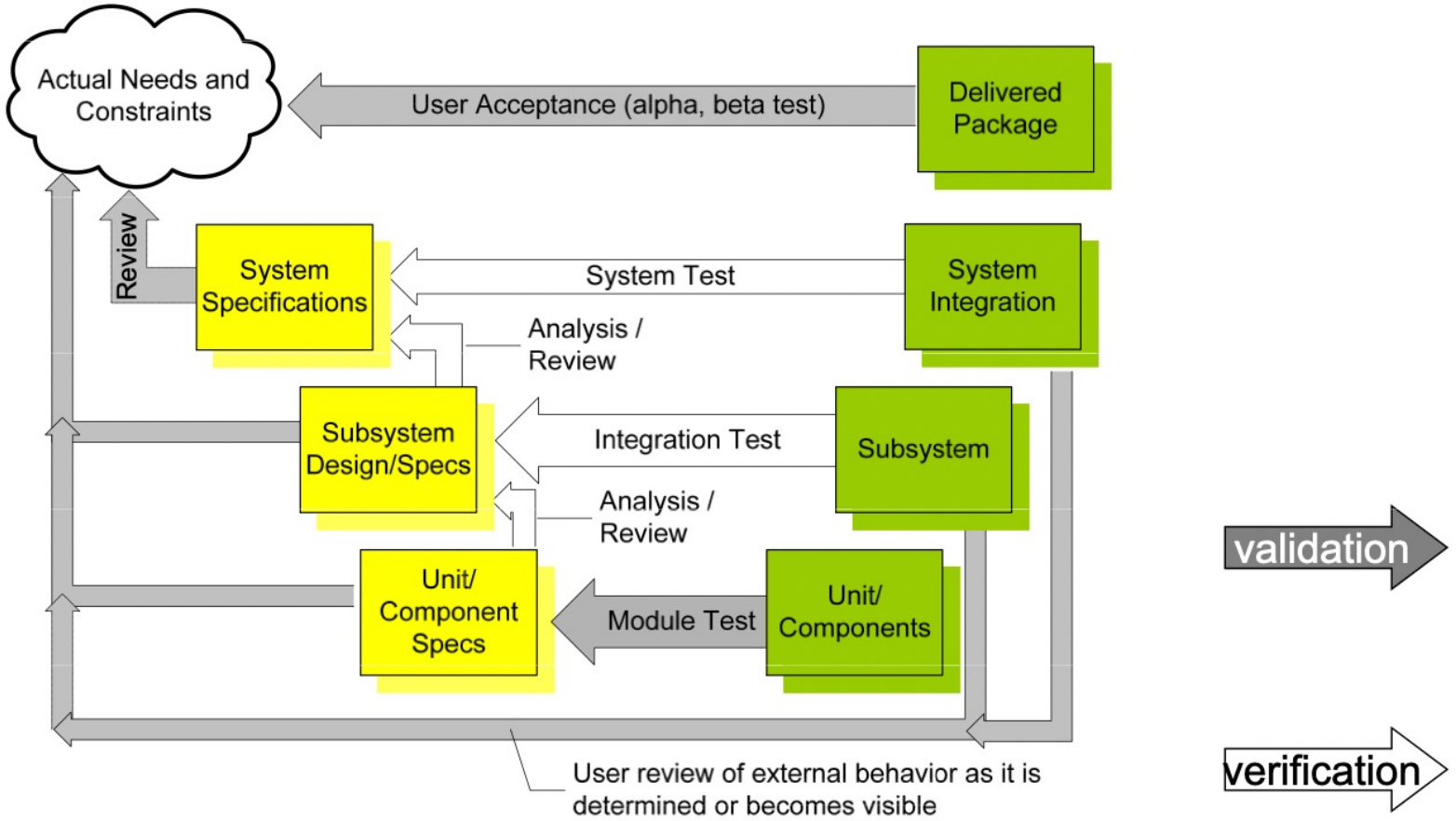
# LA VERIFICA DIPENDE DALLE SPECIFICHE




Example: elevator response

- Specifica non verificabile (ma validabile): ... se un utente preme un pulsante di richiesta al piano i, un ascensore disponibile deve arrivare il prima possibile...
- Specifica verificabile: ... se un utente preme un pulsante di richiesta al piano i, un ascensore disponibile deve arrivare al piano i entro 30 secondi...

# VERIFICA VS CONVALIDA



# TERMINOLOGIA IEEE: MALFUNZIONAMENTO

## Malfunzionamento

- Il sistema software a tempo di esecuzione non si comporta secondo le specifiche
  - Es. output non atteso
    - un malfunzionamento ha una **natura dinamica**: può essere osservato solo mediante esecuzione.
- causato da un difetto (o più difetti)

# TERMINOLOGIA IEEE: DIFETTO

Difetto (o anomalia, bug, o fault)

- è un difetto nel codice (appartiene alla struttura statica del programma)
  - l'atto di correzione dagli difetti è detto debug o bugfixing

- Normalmente causa un malfunzionamento, ma non sempre, in questo caso si dice che il **difetto** è **latente**
  - Ad esempio, il caso in cui il difetto è contenuto in un cammino che non viene praticamente mai eseguito;
  - un altro caso è rappresentato dalla presenza di più difetti il cui effetto totale è nullo

# ESEMPIO

```
// raddoppia(x) restituisce 2x
int raddoppia (int x){
     return x*x ;
     }
```

Con input 3, restituisce 9

- malfunzionamento del metodo raddoppia

Il malfunzionamento è causato dalla presenza di un difetto:

- In questo caso l'operatore * invece di +

Se testassi solo con 4, il difetto rimarebbe latente

# TERMINOLOGIA IEEE: ERRORE

## Errore

- E' la causa di un difetto
  - incomprensione umana nel tentativo di comprendere o risolvere un problema, o nell'uso di strumenti.
- Esempio: metodo raddoppia, se c'è un difetto è errore di editing (si spera...)

**Voli Usa, dietro il blocco c'è «l'errore di un ingegnere: ha sostituito un file con un altro»**

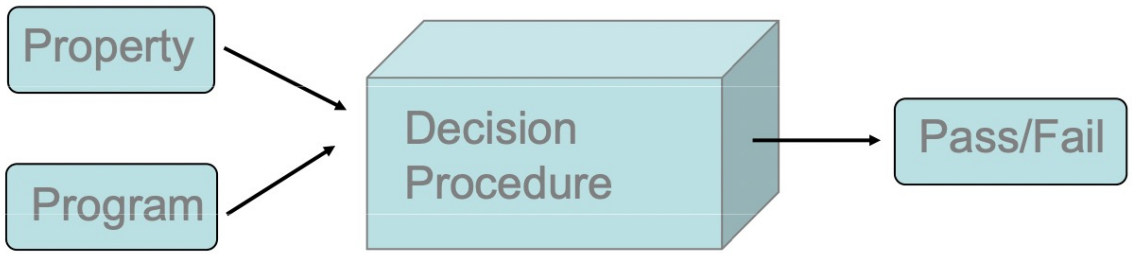
*di Leonard Berberi*

Lo stop a «Notam» ha causato la cancellazione di 1.300 voli e il ritardo di altri 10 mila

■ Cos'è il sistema andato in tilt

# IL PROBLEMA PRINCIPALE È L'INDECIDIBILITÀ DELLA CORRETTEZZA. DI UNA PROPRIETA'

You can't ~~always~~ *ever* get what you want




**Correctness properties are undecidable**

the halting problem can be embedded in almost every property of interest

# LIMITI DEL TESTING

## Getting what you need ...

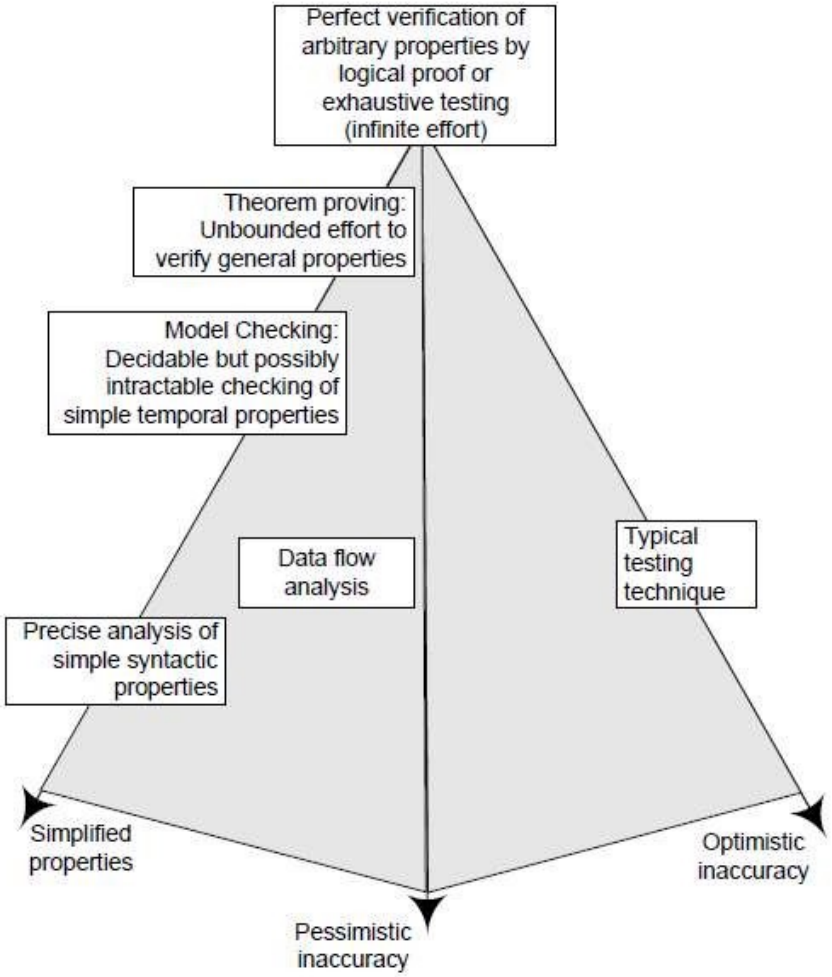
Inaccuratezza ottimistica: potremmo accettare alcuni programmi che non possiedono la proprietà (ad esempio, potrebbe non rilevare tutte le violazioni).

- testing

Inaccuratezza pessimistica: non è garantito accettare un programma anche se il programma possiede la proprietà analizzata

- tecniche di analisi automatizzata del programma

Proprietà semplificate: ridurre il grado di libertà per semplificare la proprietà da controllare



# LIMITI TEORICI E PRATICI DEL TESTING

Il Testing è una tecnica di verifica ed è come le altre sottoposta al problema dell'indecidibilità

una **prova formale di correttezza** corrisponderebbe all'esecuzione del sistema con tutti i possibili input

- testing esaustivo : eseguire e provare ogni possibile input del programma

# TESTING ESAUSTIVO

Il testing esaustivo richiederebbe:

- un tempo infinito, se gli input sono infiniti (oltre ad esserci in questi casi limiti fisici di memoria)
- un tempo troppo lungo, per domini di input finiti ma molto grandi per un programma che fa la somma di due int ci vorrebbero $2^{32} \times 2^{32} = 2^{64} \approx 10^{21}$

Test. Ipotizzando 1 nanosecondo per ogni esecuzione $10^{21} \times 10^{-9} = 10^{12} \approx 30.000$ anni

# TESI DI DIJKSTRA

- Il test di un programma può rilevare la presenza di difetti, ma non dimostrarne l'assenza

# TECNICHE DI VERIFICA: VERIFICA STATICA

## Verifica che non prevede l'esecuzione del programma

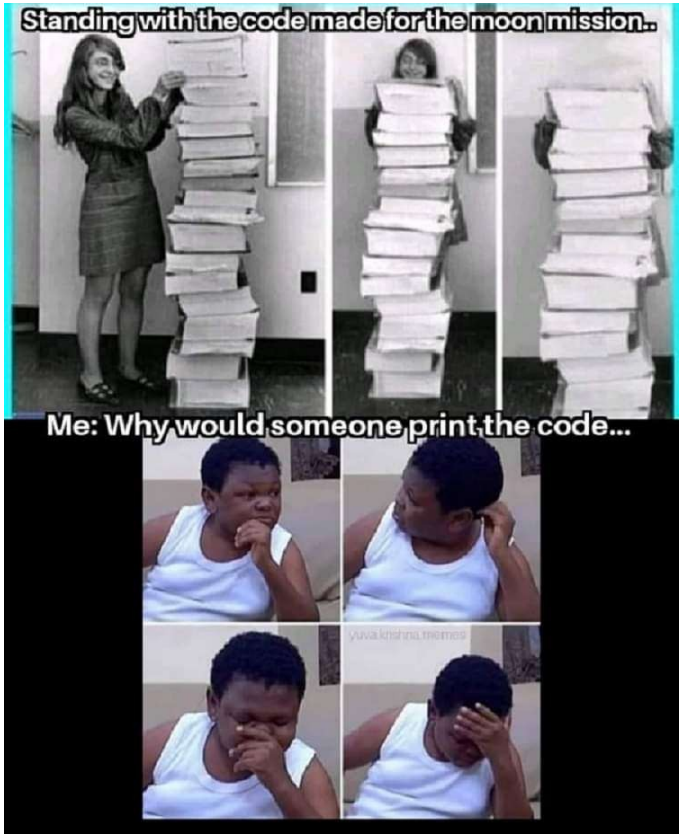
Metodi manuali

- basati sulla lettura del codice (desk-check)
- più comunemente usati
- più o meno formalmente documentati

Metodi formali o Analisi Statica supportata da strumenti

- model checking
- esecuzione simbolica
- interpretazione astratta
- theorem proving

# ORIGINE DEL DESK-CHECK





Se si voleva davvero vedere o leggere un programma prima di allora, lo si stampava

Libri e riviste, fino agli anni '70, includevano comunemente elenchi di codici. Ci si aspettava che si digitasse il programma dal listato


# 2 METODI DI LETTURA DEL CODICE

- Inspection e Walkthrough, sono metodi pratici
  - basati sulla lettura del codice
  - dipendenti dall'esperienza dei verificatori
  - per organizzare le attività di verifica
  - per documentare l'attività e i suoi risultati
- Sono metodi complementari tra loro

# INSPECTION

Si esegue una lettura mirata del codice (guidata da una lista di controllo)

Obiettivi

- rivelare la presenza di difetti

Strategia

- focalizzare la ricerca su aspetti ben definiti (error guessing)
  Ex: off-by-one error (aka Obi-Wan error)

Agenti

- verificatori diversi dai programmatori

# INSPECTION, 4 FASI

Fase 1: pianificazione

Fase 2: definizione della lista di controllo

Fase 3: lettura del codice

Fase 4: correzione dei difetti

# LISTE DI CONTROLLO

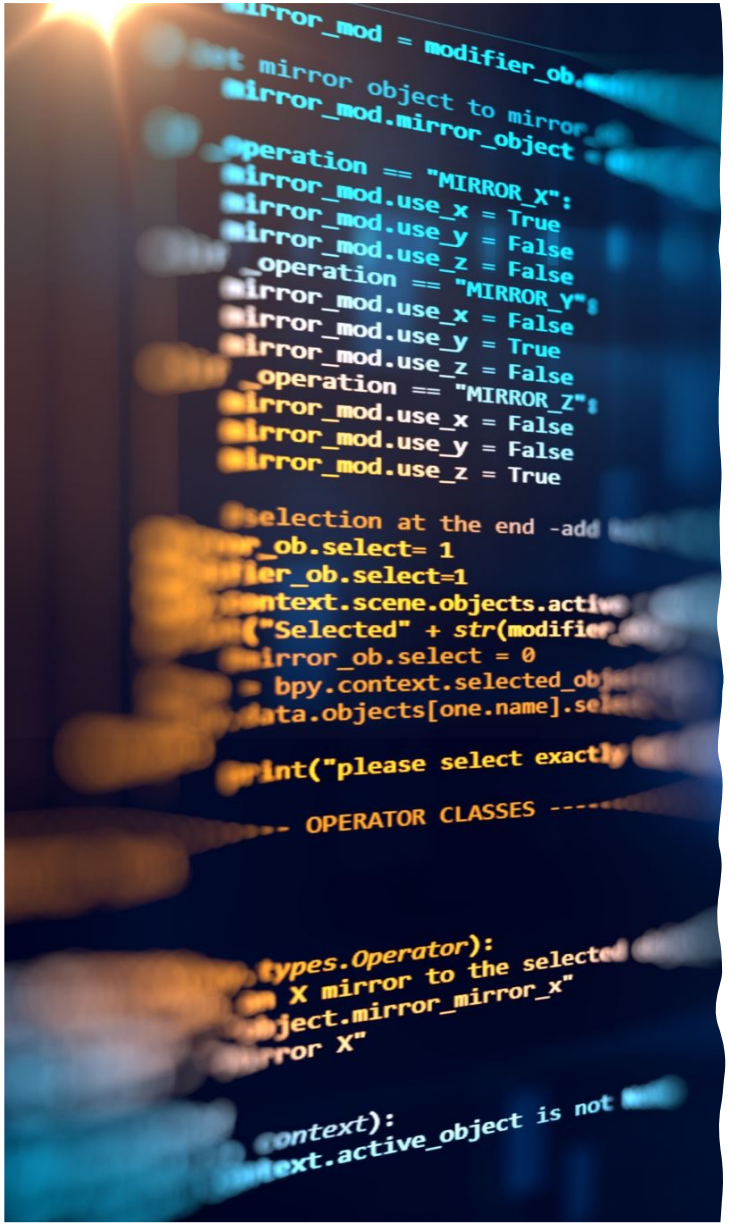
Sono frutto dell'esperienza degli ispettori

Contengono tipicamente aspetti che non possono essere controllati in maniera automatica

Le liste di controllo sono aggiornate ad ogni iterazione di inspection

# LISTE DI CONTROLLO, ESEMPI

- È stato impedito a tutti gli indici di array (o di altri insiemi) di andare fuori dai limiti?
- L'aritmetica dei numeri interi, in particolare la divisione, è utilizzata in modo appropriato?
- Tutti i file sono chiusi correttamente, anche in caso di errore?
- Tutti i riferimenti agli oggetti sono inizializzati prima dell'uso?
- Tutti gli oggetti (comprese le stringhe) sono confrontati con "equals" e non con "=="?




# LISTE DI CONTROLLO

- Errori dei dati :
  - Tutte le variabili del programma sono inizializzate prima che i loro valori vengano utilizzati?
  - A tutte le costanti è stato assegnato un nome?
  - C'è la possibilità di un buffer overflow? ecc.
- Errori di controllo:
  - La condizione è corretta per ogni istruzione condizionale?
  - E' certo che ogni ciclo termina?
  - Le istruzioni composte sono o non sono state messe tra parentesi in modo corretto?
- Errori di ingresso/uscita (I/O) :
  - Tutte le variabili di ingresso sono utilizzate o no?
  - Tutte le variabili di uscita sono istanziate prima di essere restituite?
  - Gli input inattesi possono essere causa di fault? ecc.


# LISTE DI CONTROLLO

- Difetti di interfaccia:
  - Tutti i metodi e le funzioni hanno il numero corretto di parametri?
  - Il tipo di parametri, cioè effettivi e formali, corrisponde?
  - I parametri sono presenti nell'ordine corretto?
  - Se tutti i componenti accedono a una memoria condivisa , hanno lo stesso modello della struttura della memoria condivisa?
- Errori nella gestione della memoria:
  - Se si utilizza lo storage dinamico, lo spazio è stato allocato correttamente?
  - Lo spazio viene deallocato esplicitamente dopo che non è più necessario? ecc.
- Gestione delle eccezioni:
  - Sono state prese in considerazione tutte le possibili condizioni di errore?

## Java Checklist: Level 1 inspection (single-pass read-through, context independent)

FEATURES (where to look and how to check):
Item (what to check)

| *FILE HEADER: Are the following items included and consistent?* | yes | no | comments |
|---|---|---|---|
| Author and current maintainer identity | | | |
| Cross-reference to design entity | | | |
| Overview of package structure, if the class is the principal entry point of a package | | | |
| *FILE FOOTER: Does it include the following items?* | yes | no | comments |
| Revision log to minimum of 1 year or at least to most recent point release, whichever is longer | | | |
| *IMPORT SECTION: Are the following requirements satisfied?* | yes | no | comments |
| Brief comment on each import with the exception of standard set: java.io.*, java.util.* | | | |
| Each imported package corresponds to a dependence in the design documentation | | | |
| *CLASS DECLARATION: Are the following requirements satisfied?* | yes | no | comments |
| The visibility marker matches the design document | | | |
| The constructor is explicit (if the class is not *static*) | | | |
| The visibility of the class is consistent with the design document | | | |
| *CLASS DECLARATION JAVADOC: Does the Javadoc header include:* | yes | no | comments |
| One sentence summary of class functionality | | | |
| Guaranteed invariants (for data structure classes) | | | |
| Usage instructions | | | |
| *CLASS: Are names compliant with the following rules?* | yes | no | comments |
| Class or interface: CapitalizedWithEachInternalWordCapitalized<br>Special case: If class and interface have same base name, distinguish as ClassNameIfc and ClassNameImpl | | | |
| Exception: ClassNameEndsWithException | | | |
| Constants (final): ALL_CAPS_WITH_UNDERSCORES | | | |
| Field name: capsAfterFirstWord.<br>name must be meaningful outside of context | | | |
| *IDIOMATIC METHODS: Are names compliant with the following rules?* | yes | no | comments |
| Method name: capsAfterFirstWord<br>Local variables: capsAfterFirstWord.<br>Name may be short (e.g., i for an integer) if scope of declaration and use is less than 30 lines. | | | |
| Factory method for X: newX | | | |
| Converter to X: toX | | | |
| Getter for attribute x: getX(); | | | |
| Setter for attribute x: void setX | | | |

## Java Checklist: Level 2 inspection (comprehensive review in context)

FEATURES (where to look and how to check):
Item (what to check)

| *DATA STRUCTURE CLASSES: Are the following requirements satisfied?* | yes | no | comments |
|---|---|---|---|
| The class keeps a design secret | | | |
| The substitution principle is respected: Instance of class can be used in any context allowing instance of superclass or interface | | | |
| Methods are correctly classified as constructors, modifiers, and observers | | | |
| There is an abstract model for understanding behavior | | | |
| The structural invariants are documented | | | |
| *FUNCTIONAL (STATELESS) CLASSES: Are the following requirements satisfied?* | yes | no | comments |
| The substitution principle is respected: Instance of class can be used in any context allowing instance of superclass or interface | | | |
| *METHODS: Are the following requirements satisfied?* | yes | no | comments |
| The method semantics are consistent with similarly named methods. For example, a "put" method should be semantically consistent with "put" methods in standard data structure libraries | | | |
| Usage examples are provided for nontrivial methods | | | |
| *FIELDS: Are the following requirements satisfied?* | yes | no | comments |
| The field is necessary (cannot be a method-local variable) | | | |
| Visibility is protected or private, or there is an adequate and documented rationale for public access | | | |
| Comment describes the purpose and interpretation of the field | | | |
| Any constraints or invariants are documented in either field or class comment header | | | |
| *DESIGN DECISIONS: Are the following requirements satisfied?* | yes | no | comments |
| Each design decision is hidden in one class or a minimum number of closely related and co-located classes | | | |
| Classes encapsulating a design decision do not unnecessarily depend on other design decisions | | | |
| Adequate usage examples are provided, particularly of idiomatic sequences of method calls | | | |
| Design patterns are used and referenced where appropriate | | | |
| If a pattern is referenced: The code corresponds to the documented pattern | | | |

# DATI SULL'EFFICACIA DELL'INSPECTION

Raytheon

- Riduzione del " rework " dal 41% del costo al 20% del costo
- Riduzione dell'80% dello sforzo per risolvere i problemi di integrazione-

Paulk et al.: costo per la correzione di un difetto nel software dello Space Shuttle

- 1 $ se trovato durante l'ispezione
- 13 $ durante il test del sistema
- 92 $ dopo la consegna

IBM

- 1 ora di ispezione ha risparmiato 20 ore di test
- Risparmio di 82 ore di rilavorazione in caso di difetti nel prodotto rilasciato-

Laboratorio IBM di Santa Teresa

- 3,5 ore per trovare un bug con l'ispezione, 15-25 con il test

C. Jones

- Le ispezioni di progettazione/codice eliminano il 50-70% dei difetti
- I test eliminano il 35%.

R. Grady

- Uso del sistema 0,21 difetti/ora
- Scatola nera 0,28 difetti/ora
- Scatola bianca 0,32 difetti/ora
- Lettura/ispezione 1,06 difetti/ora

# WALKTHROUGH

## Obiettivo

- rivelare la presenza di difetti
- eseguire una lettura critica del codice

## Strategia

- percorrere il codice simulandone l'esecuzione

## Agenti

- gruppi misti ispettori e sviluppatori

# LE FASI DEL WALKTHROUGH

FASE 1: PIANIFICAZIONE

FASE 2: LETTURA DEL CODICE

FASE 3: CORREZIONE DEI DIFETTI

# VANTAGGI DELLA VERIFICA MANUALE (DESK-CHECK)

Praticità e intuitività

Ideale per alcune caratteristiche di qualità

Convenienza economica

costi dipendenti dalle dimensioni del codice

bassi costi di infrastruttura

buona prevedibilità dei risultati

# INSPECTION VS WALKTHROUGH

## Affinità

- controlli statici basati su desk-test
- programmatori e verificatori contrapposti
- documentazione formale

## Differenze

- inspection basato su errori presupposti
- walkthrough è più completo
- inspection più rapido

# METODI FORMALI

Tecnica basata sulla dimostrazione formale di correttezza di un modello finito (dimostrazione possibile) e istanziazione del modello.

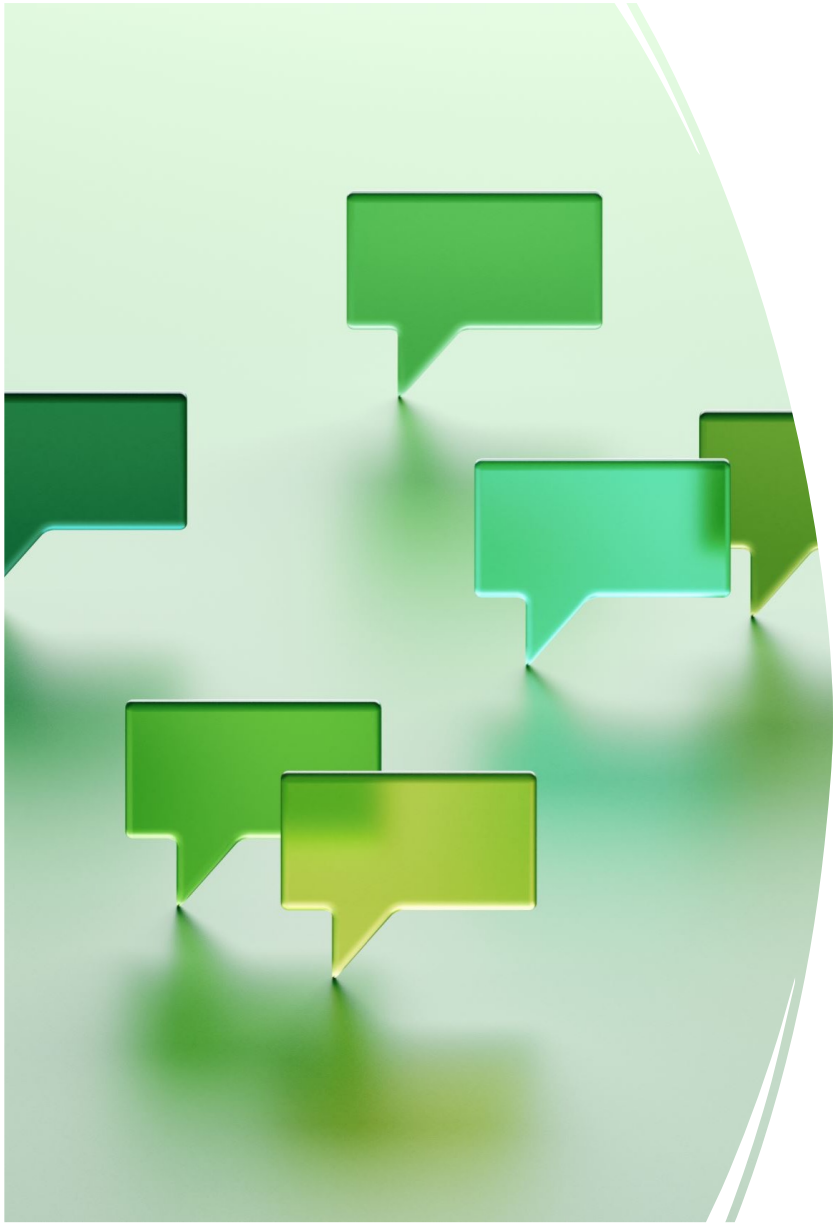
Esempio protocollo «two-phase locking»:

si dimostra corretto

se istanziato correttamente garantisce assenza di malfunzionamenti dovuti alla race condition

# METODI FORMALI

- Il two-phase locking (2PL), locking a due fasi, è un protocollo per il controllo della concorrenza utilizzato nel campo dei database e dell'elaborazione delle transazioni per garantire la serializzabilità di una schedule di transazioni.
- Questo protocollo è basato sui lock, applicabili da una transazione ad un dato (a diversi livelli di granularità: attributo, tupla, relazione, database), i quali bloccano l'accesso da parte di altre transazioni ai dati fino alla terminazione della transazione che li detiene.



# METODI FORMALI

- Osservazioni (es: two-phase locking):
  - Ci sono applicazioni che non usano «two-phase locking» e sono corrette
  - Occorre comunque provare che il programma applica correttamente il protocollo, ma di solito è più facile che provare l'assenza di malfunzionamenti in generale

# METODI FORMALI: TRIPLE DI HOARE

- La logica di Hoare è un sistema formale che rientra tra le semantiche assiomatiche
- pubblicato per la prima volta nel 1969 da C. A. R. Hoare
- Definisce un insieme iniziale di assiomi e di regole su di essi, di valutare la correttezza di programmi utilizzando il rigore dei formalismi matematici.
- La logica è stata sviluppata per essere utilizzata con un semplice linguaggio di programmazione imperativo ed ha subito sviluppi ulteriori per merito dello stesso Hoare e di altri ricercatori per la gestione di casistiche particolari quali la concorrenza, i puntatori e le procedure.

# METODI FORMALI: TRIPLE DI HOARE

Questa tripla definisce come l'esecuzione di un comando modifichi la verità in merito al programma, ed è definita come

`{P}C{Q}`

- In cui *P* e *Q* vengono chiamate asserzioni e *C* è un comando. Nel caso specifico *P* viene definita precondizione e *Q* postcondizione.
- Nel suo documento originale Hoare definisce il significato della tripla come:

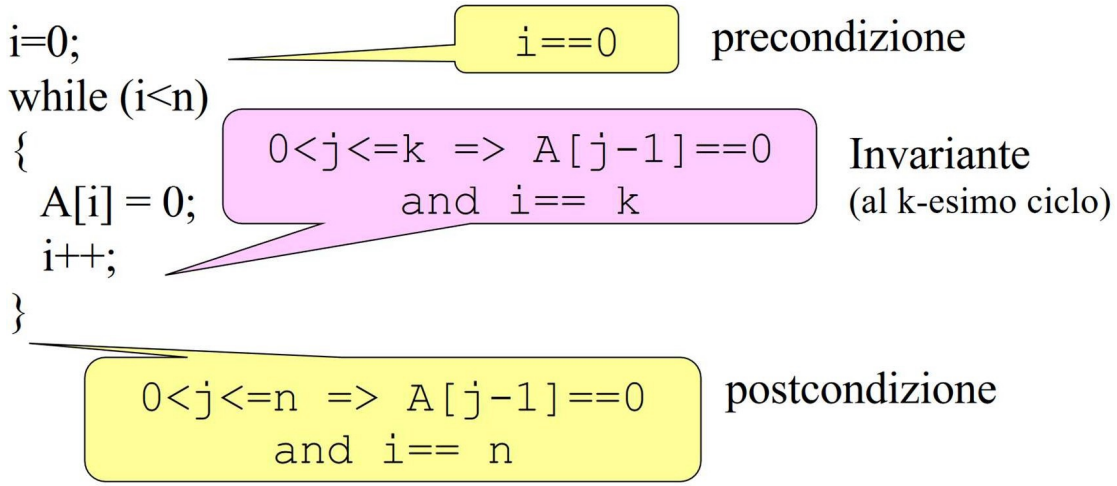
«Se l'asserzione *P* è vera prima dell'esecuzione di un comando o di una serie di comandi *C*, allora *Q* è vera dopo l'esecuzione.»

- Nel caso in cui non vi siano precondizioni da rispettare scriviamo semplicemente

`{true}C{Q}`

- Il sistema per verificare le triple utilizza assiomi, ossia triple che risultano sempre soddisfatte, e regole di inferenza che permettono di semplificare il comando per induzione strutturale.

# METODI FORMALI: TRIPLE DI HOARE



# B METHOD

La notazione B dipende dalla teoria degli insiemi e dalla logica del primo ordine per specificare diverse versioni del software che coprono il ciclo completo di sviluppo del progetto.

## Macchina astratta

Nella prima e più astratta versione, che è chiamata Macchina astratta, il progettista dovrebbe specificare l'obiettivo del progetto.

## Perfezionamento

Quindi, durante una fase di perfezionamento, possono riempire la specifica per chiarire l'obiettivo o per rendere la macchina astratta più concreta aggiungendo dettagli sulle strutture dati e sugli algoritmi che definiscono come viene raggiunto l'obiettivo.

Il progettista può utilizzare librerie B per modellare strutture dati o per includere o importare componenti esistenti.

## Implementazione

Il perfezionamento continua fino a quando non viene ottenuta una versione deterministica: l'implementazione.

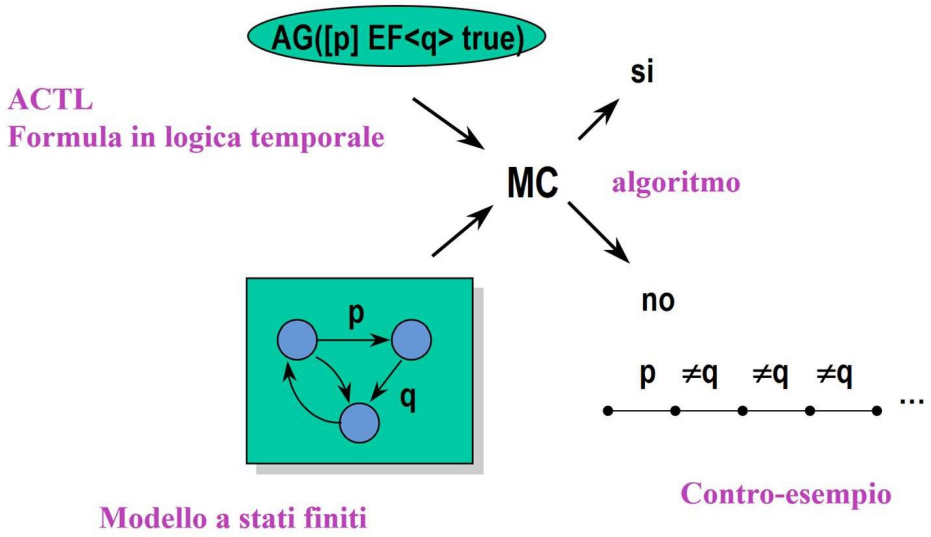
Durante tutte le fasi di sviluppo viene utilizzata la stessa notazione e l'ultima versione può essere tradotta in un linguaggio di programmazione per la compilazione.

# METODI FORMALI: B METHOD


L'applicazione più nota del B method è la metropolitana automatica METEOR, linea 14 di Parigi

# Metodi Formali: Model Checking

(Clarke/Emerson,Queille/Sifakis) - 1986




**Il modello deve rappresentare tutti i comportamenti**

# METODI FORMALI: MODEL CHECKING

- Formalmente il problema è posto così: scelta una proprietà da verificare, espressa come una formula logica temporale p, e un modello M avente stato iniziale s, decidere se

$$M, s \models p.$$

- Gli strumenti del model checking si scontrano con la crescita esponenziale dell'insieme degli stati, comunemente conosciuto come il problema dell'esplosione combinatoria, che deve servire a risolvere la maggior parte dei problemi del mondo reale.
- I ricercatori hanno sviluppato algoritmi simbolici, riduzione parziale dell'ordine, diagrammi decisionali, astrazioni e model checking al volo per risolvere il problema. Questi strumenti furono inizialmente sviluppati per la correttezza logica dei sistemi a stati discreti, ma da allora sono stati estesi per trattare sistemi sistema real-time e forme limitate di sistemi ibridi.

# RIASSUMENDO...

- La maggior parte delle proprietà interessanti sono indecidibili, quindi in generale non possiamo contare su strumenti che funzionino senza l'intervento umano
- • La valutazione delle qualità del programma comprende due serie complementari di attività: convalida (il software fa ciò che dovrebbe fare?) e verifica (il sistema si comporta come specificato?)
- • Non esiste una tecnica unica per tutti gli scopi: i progettisti dei test devono selezionare una combinazione adatta di tecniche

# LETTURA CONSIGLIATA

- Cap 1-2 -18

  Software Testing and Analysis: Process, Principles and Techniques- Mauro Pezzè e Michal Young