18. Verifica e validazione

IS 2024-2025

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

LASCIATE OGNI SPERANZA O VOI CHE ENTRATE

Il problema della verifica di correttezza è molto difficile
(o meglio **indecidibile**, i.e. non esiste un algoritmo che lo risolva)

UN RISULTATO FONDAMENTALE

Esistono problemi **non decidibili**, per i quali cioè non esiste alcun algoritmo in grado di dare una risposta in tempo finito su tutte le istanze del problema (A. Turing, 1937)

Coinvolgono il **problema della terminazione**:

- stabilire, ricevuti in ingresso un qualsiasi programma e un suo possibile input, se l'esecuzione di tale programma sullo specifico input termina (oppure no)

PROBLEMA DELLA TERMINAZIONE

Assumiamo che esista un programma `P` che prende in input un altro programma, chiamato `Q`, e un input `D` per il programma `Q`. Il programma `C` verifica se `Q` termina su `D` come segue:

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

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

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

Il programma `K` con input `K` termina?

PROBLEMA DELLA TERMINAZIONE (CONT.)

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

Il programma `K` con input `K` termina?

- `K` con input `K` termina solo se si entra nel ramo `else` (restituendo il valore `false`)
- Questo avviene se e solo se `P(K,K)` è `false`
- `P(K,K)` è falso solo se `halts(K(K))` è falso, ovvero se `K` con input `K` **non** termina

In breve, `K(K)` termina se e solo se `K(K)` non termina ⇒ **contraddizione**

- Non esiste un programma `P` che per ogni programma `Q` e input `P`, dice se il programma `Q` sull'input `D` termina o no (aka. **halting problem**)

UN RISULTATO FONDAMENTALE (CONT.)

Non esiste un programma P che per ogni programma Q e input P, dice se il programma Q sull'input D termina o no (aka. **halting problem**)

Non è solo un risultato teorico

- Molte proprietà interessanti del software incorporano il problema della terminazione
- 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

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

Esistono problemi **non decidibili**, per i quali cioè non esiste alcun algoritmo in grado di dare una risposta in tempo finito su tutte le istanze del problema (A. Turing, 1937)

UN ALTRO ESEMPIO DI INDECIDIBILITÀ

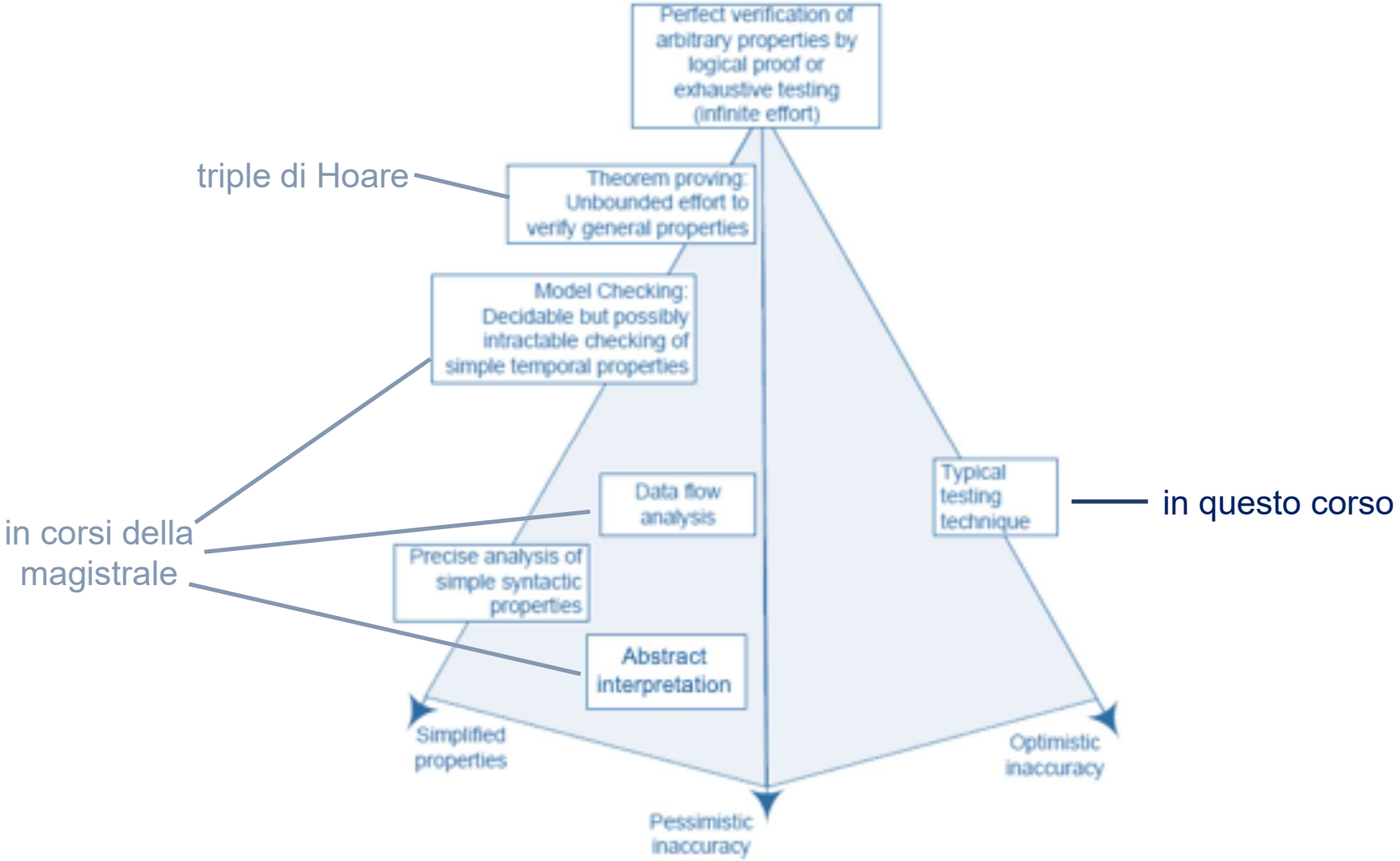
Dire se due programmi producono lo **stesso risultato**, dati gli stessi input

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");
}
```

VERIFICA E VALIDAZIONE: COSA SI PUÒ FARE?



TRIPLE DI HOARE

Perché non ci bastano? Dove si nasconde il problema?

↓

La logica del primo ordine è **indecidibile**

- Data una formula F scritta con la logica del primo ordine, esiste un algoritmo per decidere in tempo finito se F è verificata? ($\vDash F$ o $\nvDash F$?)

- No, non esiste

- Si possono enumerare le formule valide, ma (in **tempo finito**) non è detto che si dimostri F o $\neg F$

VERIFICA E VALIDAZIONE (V&V)

V&V come attività di **software quality assurance**

- Parte integrante del processo software, da svolgere in **ogni fase**

- Per confermare che **processo** e **prodotto** rispettino i **requisiti di qualità**

- Citando Barry Bohem, V&V consentono di rispondere alle domande
    - «*Are we building the **product right**?*»
    - «*Are we building the **right product**?*»

V&V: DIFFICOLTÀ DA CONSIDERARE

- I **requisiti** di **qualità** sono **diversi** tra loro
- Un prodotto software è in costante **evoluzione**
- I guasti hanno una **distribuzione irregolare**
- V&V sono **non-lineari**
  - se un ascensore può trasportare 1000kg, allora può trasportare anche carichi inferiori
  - se una procedura ordina correttamente X elementi, non è detto che ordini correttamente anche Y elementi (con $Y \neq X$)
- **Approcci diversi** possono introdurre **errori diversi**
  - 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 adatti a
  - **problema**,
  - **requisiti** e
  - **ambiente di sviluppo**

(senza poter contare su «ricette» fisse)

CINQUE DOMANDE DA USARE COME GUIDA

1. Quando **iniziare** verifica e convalida? Quando sono **complete**?

2. Quali **tecniche** applicare?

3. Come valutare se un **prodotto** è **pronto** per essere rilasciato?

4. Come possiamo controllare la qualità delle **release successive**?

5. Come può essere migliorato il **processo** di sviluppo?

1) QUANDO INIZIARE? QUANDO FINIRE?

Il testing **NON è una fase finale** dello sviluppo software

- L'esecuzione di test è solo una piccola parte del processo di V&V
- V&V **iniziano** quando decidiamo di **creare** un prodotto software
- V&V **continuano dopo** la **consegna** di un prodotto software
  - Finché il prodotto viene usato
  - Per far fronte alle evoluzioni e agli adattamenti a nuove condizioni

1) QUANDO INIZIARE? QUANDO FINIRE? (CONT.)

V&V devono essere avviate dallo **studio di fattibilità**

- Tenere conto di **qualità richieste** e **impatto sul costo** complessivo
- Svolgere **attività correlate alla qualità**, ivi comprese
    - analisi del rischio
    - definizione delle misure 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

1) QUANDO INIZIARE? QUANDO FINIRE? (CONT.)

V&V continuano **dopo il rilascio**

- Attività di **manutenzione**, ivi comprese
    - analisi delle modifiche ed estensioni
    - generazione di nuove suite di test per le funzionalità aggiuntive
    - ripetizione 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?

Non basta una singola tecnica per tutti gli scopi di analisi e testing (A&T)

Meglio **combinare** diverse tecniche di A&T

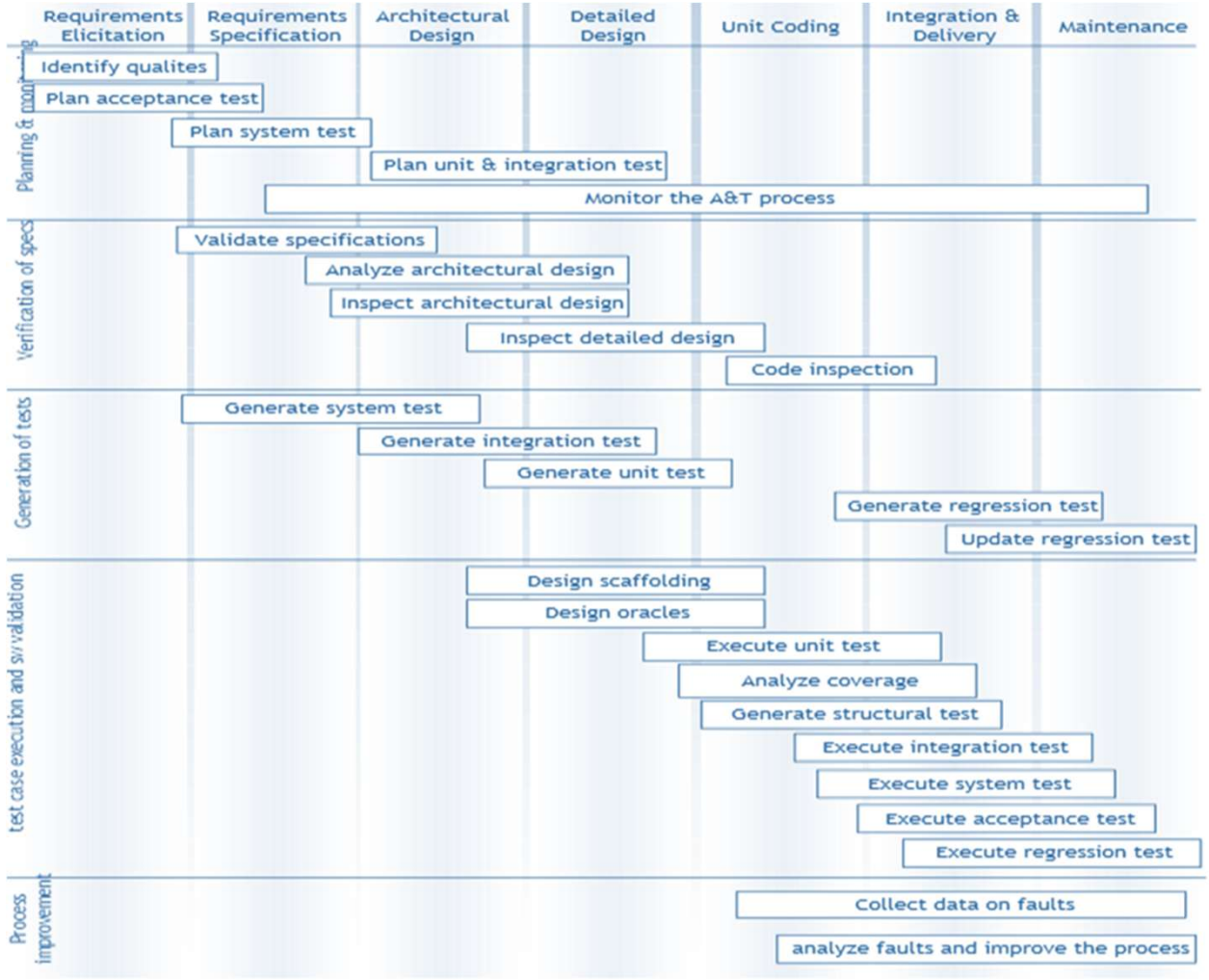
- **Efficacia diversa** per diverse classi di difetti (p.e. l'analisi statica è meglio del testing per l'identificazione di race condition)
- **Applicabilità** in **diverse fasi** del processo di sviluppo (p.e. ispezione per la convalida dei requisiti iniziali, testing per la validazione del prodotto sviluppato)
- Differenze negli **scopi** (p.e. test statistico per misurare l'affidabilità)
- **Compromessi** in termini di **costo** e **affidabilità** (p.e. usare tecniche costose solo per requisiti di sicurezza)

2) QUALI TECNICHE APPLICARE? (CONT.)

tecniche diverse

in

fasi diverse



3) COME VALUTARE PRODOTTO PRONTO?

- Individuare le **misure** di **qualità** del software di interesse
  - **disponibilità**, in termini di tempo di esecuzione vs. tempo in cui il sistema è giù
  - **tempo medio tra i guasti**, ovvero il tempo medio tra un guasto e il successivo
  - **affidabilità**, misurata come la percentuale di operazioni che terminano con successo

- **Definire bene** tali **misure**
  - Applicazione web con 100 operazioni
  - Tutte le operazioni funzionano bene, tranne una che «sbaglia» nel 50% dei casi
  - Qual è l'affidabilità del sistema?
    - Se contiamo la percentuale di operazioni corrette: 99%
    - Se contiamo le sessioni corrette e chiamiamo solo l'operazione «sbagliata»: 50%

3) COME VALUTARE PRODOTTO PRONTO? (CONT.)

Alfa test

- Eseguiti da sviluppatori o utenti selezionati
- In ambiente controllato
- Osservati dall'organizzazione dello sviluppo

Beta test

- Eseguiti da utenti reali
- Nel loro ambiente
- Attività reali senza interferenze o monitoraggio ravvicinato

4) COME CONTROLLARE RELEASE SUCCESSIVE?

Attività **dopo** la consegna

- test e analisi di **codice nuovo** o **modificato**
- **ripetizione** dei **test** di sistema
- **memorizzazione** di tutti i **bug** trovati
- test di **regressione**
- **distinzione** tra cambiamenti di **versione** «major» e «minor» (p.e. 2.0 o 1.5, dopo la 1.4)

5) COME MIGLIORARE IL PROCESSO?

Si incontrano gli stessi difetti progetto dopo progetto

- identificare e rimuovere i **punti deboli** nel **processo** di sviluppo
  (p.e. cattive pratiche di programmazione)

- identificare e rimuovere i **punti deboli** del **test** e dell'**analisi**
  (che non permettono di individuare i punti deboli di cui sopra)

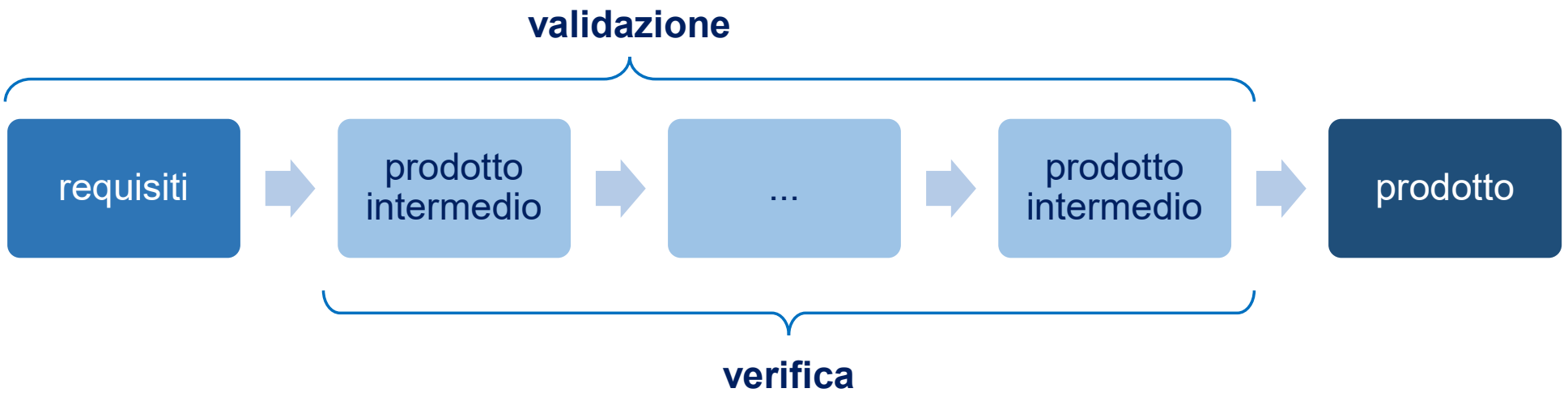
VALIDAZIONE O VERIFICA?

La **convalida** (o **validazione**, dall'inglese «**validation**») risponde alla domanda:

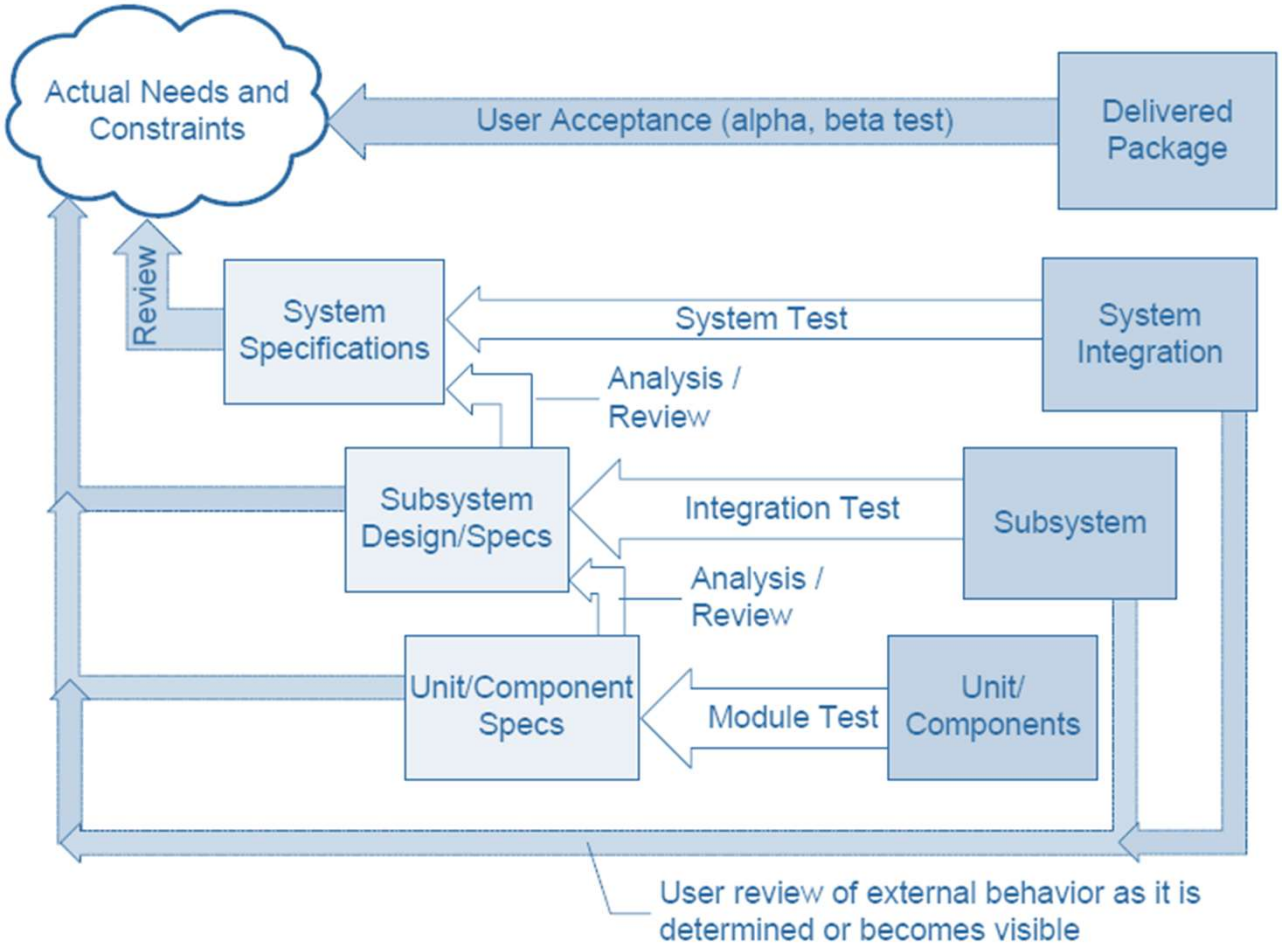
- stiamo costruendo il sistema che **serve all'utente**?

La **verifica** risponde alla domanda:

- stiamo costruendo un sistema che **rispetta le specifiche**?



VALIDAZIONE O VERIFICA? (CONT.)



UN PO' DI TERMINOLOGIA, DA IEEE

malfunzionamento = il sistema software, a runtime, non si comporta secondo le specifiche

- Ha natura **dinamica** e può essere osservato solo mediante **esecuzione** (es. output non atteso)
- È causato da un **difetto** (o più difetti)

difetto = anomalia, bug, o fault nel codice del sistema software

- Appartiene alla struttura **statica** del programma
- L'atto di correzione/risoluzione dei difetti è detto **debugging** o **bug fixing**
- Può causare un **malfunzionamento** (o più malfunzionamenti)
- Se non causa malfunzionamenti, si dice che il difetto è **latente**, ad esempio
    - Quando è contenuto in un flusso che non viene (quasi) mai eseguito
    - Quando sono presenti più difetti il cui effetto totale è nullo

ESEMPIO

```
// La funzione raddoppia(x) restituisce il doppio di x
int raddoppia(int x){
  return x*x;
}
```

Proviamolo!

`raddoppia(3)` → `9` ⇒ **malfunzionamento** del metodo `raddoppia`

Il malfunzionamento è causato dalla presenza di un **difetto**

x*x invece di x+x

NB: Se testassi solo `raddoppia(2)`, il difetto rimarrebbe latente

UN PO' DI TERMINOLOGIA, DA IEEE (CONT.)

malfunzionamento = il sistema software, a runtime, non si comporta secondo le specifiche

difetto = anomalia, bug, o fault nel codice del sistema software

errore = è la causa di un difetto

- Un errore umano nella comprensione o risoluzione di un problema (o nell'uso di strumenti)
- Ad esempio, il difetto del metodo `raddoppia`, è dovuto ad un errore di editing (si spera ☺)

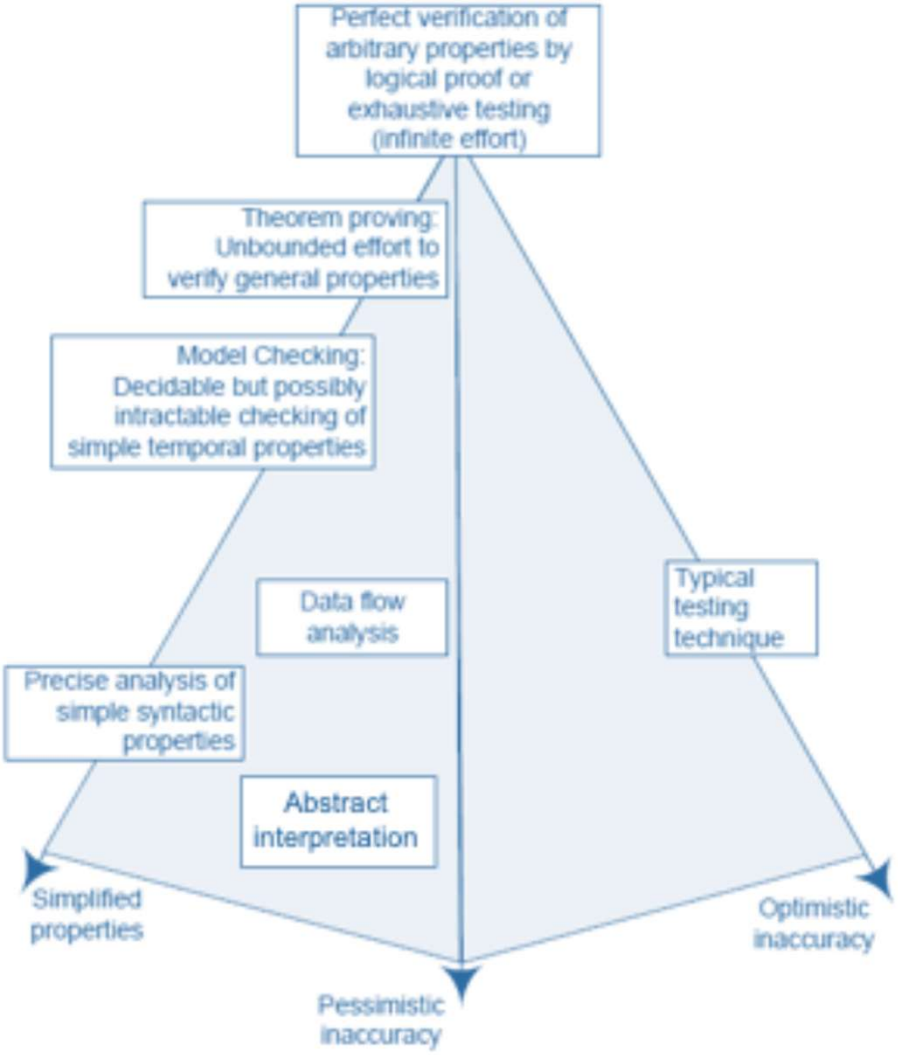
Voli Usa, dietro il blocco c'è «l'errore di un ingegnere: ha sostituito un file con un altro»

di Leonard Berberi

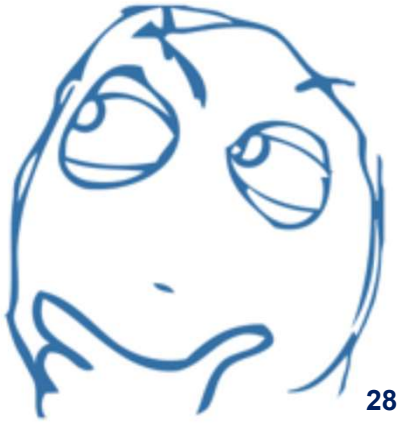
Lo stop a «Notam» ha causato la cancellazione di 1.300 voli e il ritardo di altri 10 mila

Cos'è il sistema andato in tilt

LIMITI DEL TESTING



optimistic inaccuracy? cosa significa?



LIMITI DEL TESTING (CONT.)

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

(una **prova di correttezza** corrisponderebbe all'esecuzione un sistema con **tutti i possibili input**)

Testing esaustivo = eseguire e provare **ogni possibile input** di un programma

- Richiede tempo infinito, se gli input sono infiniti (oltre ai limiti fisici della memoria per questi casi)
- Richiede troppo tempo, per domini di input finiti ma molto grandi

Ad esempio, il seguente programma Python calcola quanti anni servono per testare tutti i possibili input di un programma che fa la somma di due `unsigned int` (su 32 bit)

```
nanosecondi = 2**64
secondi = nanosecondi / (10**9)
ore = secondi / 3600
giorni = ore / 24
anni = giorni / 365
print(round(anni)) # = ???
```

585 anni?

LA TESI DI DIJSKTRA

Program testing can be used to show the presence of bugs, but never to show their absence! (E. W. Dijkstra)

Vedremo come, dopo aver parlato di verifica statica

TECNICHE DI VERIFICA STATICA

La verifica **statica** non prevede l'esecuzione del programma

Metodi **manuali**

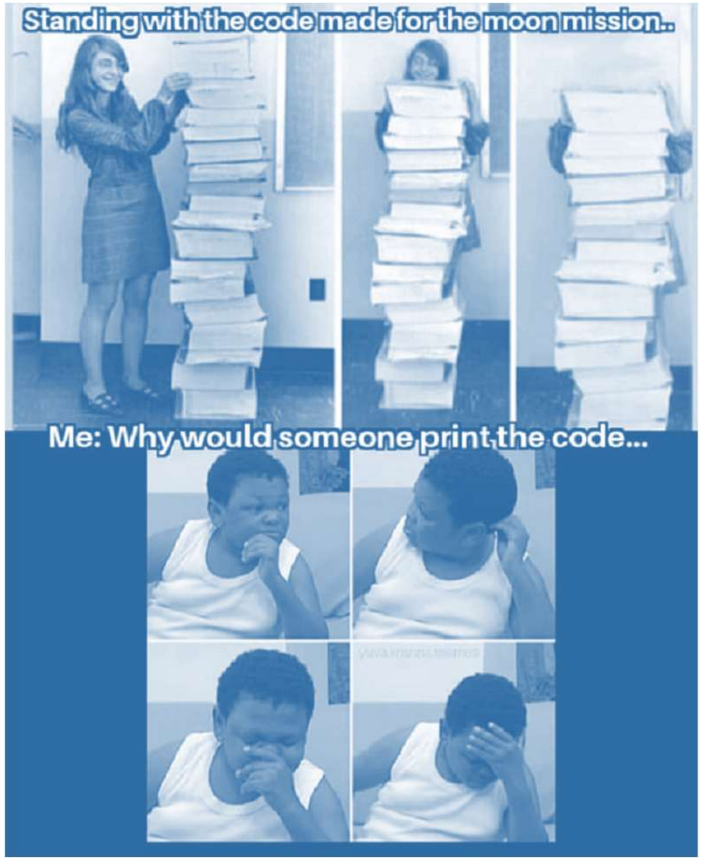
- basati sulla lettura del codice (desk-check)
- più comunemente usati
- più o meno formalmente documentati

Metodi **formali** o analisi statica supportata da **strumenti**

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

DESK-CHECK DEL CODICE

Analisi manuale di **programmi software** volta a capire **cosa facciano** e/o ad **identificare errori logici** che potrebbero occorrere durante la loro esecuzione



Origini:

- In passato, se si voleva vedere o leggere un programma, lo si stampava

- Libri e riviste, fino agli anni '70, includevano comunemente elenchi di codici
  → Ci si aspettava che si (ri)digitasse il programma

DESK-CHECK DEL CODICE (CONT.)

Due possibilità: **inspection** e **walkthrough**

- 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

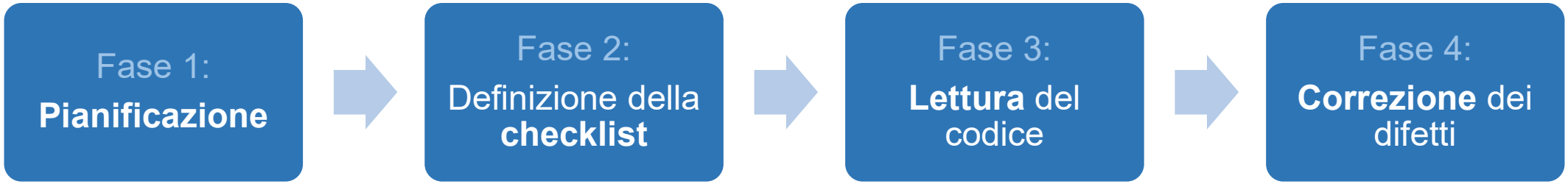
- Metodi **complementari** tra loro

DESK-CHECK MEDIANTE INSPECTION

Si esegue una **lettura mirata** del codice, guidata da una **checklist**

- **Obiettivo**: rivelare la presenza di **difetti**
- **Strategia**: focalizzare la ricerca su aspetti ben definiti (**error guessing**)
  Ad esempio, off-by-one error (aka Obi-Wan error)
- **Agenti**: **ispettori**, diversi dai programmatori

Svolta in **quattro fasi**:



DESK-CHECK MEDIANTE INSPECTION (CONT.)

Le checklist sono **frutto dell'esperienza** degli ispettori

- Tipicamente, contengono aspetti che non controllabili in maniera automatica
- Sono aggiornate ad ogni iterazione di inspection

Esempio:

- ❑ 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 `==`?

DESK-CHECK MEDIANTE INSPECTION (CONT.)

Un altro esempio di checklist:

- ❑ 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.

DESK-CHECK MEDIANTE INSPECTION (CONT.)

Un altro esempio di checklist:

- ❑ 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?

DESK-CHECK MEDIANTE INSPECTION (CONT.)

Un altro esempio di checklist, dal web:

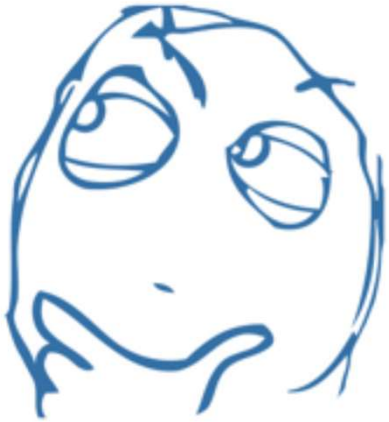
Java Checklist: Level 1 inspection (single-pass read-through, context independent)

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 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. name must be meaningful outside of context			
IDIOMATIC METHODS: Are names compliant with the following rules?	yes	no	comments
Method name: capsAfterFirstWord Local variables: capsAfterFirstWord. 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			

Ok, ma almeno è efficace?



DESK-CHECK MEDIANTE INSPECTION (CONT.)

Alcuni dati sull'**efficacia** di desk-check mediante inspection

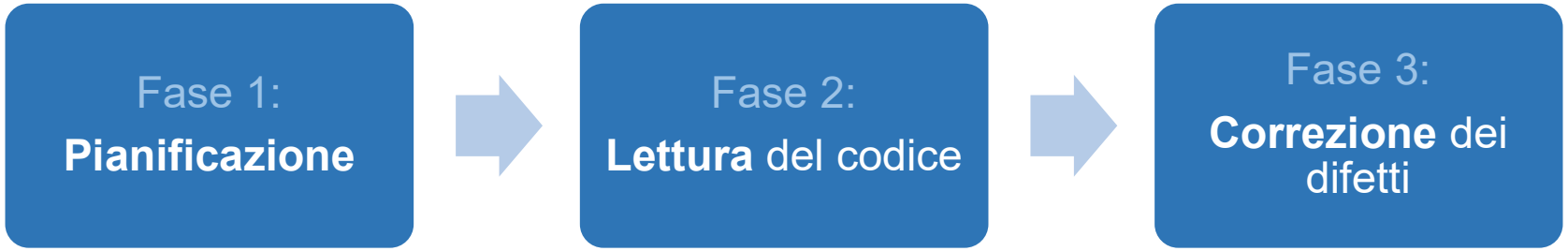
- Raytheon
  - Rework ridotto al 20% del costo (dal 41%)
  - Sforzo per risolvere i problemi di integrazione ridotto dell'80%
- Costo per la correzione di un difetto nel software dello Space Shuttle (Paulk et al.)
  - 1 $ se trovato durante l'ispezione
  - 13 $ durante il test del sistema
  - 92 $ dopo la consegna
- IBM
  - 1 ora di ispezione (contro 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

DESK-CHECK MEDIANTE WALKTHROUGH

Si esegue una **lettura critica** del codice

- **Obiettivo**: rivelare la presenza di **difetti**
- **Strategia**: percorrere il codice **simulandone l'esecuzione**
- **Agenti:** gruppi misti **ispettori** e **sviluppatori**

Svolta in **tre fasi**:



INSPECTION VS WALKTRHOUGH

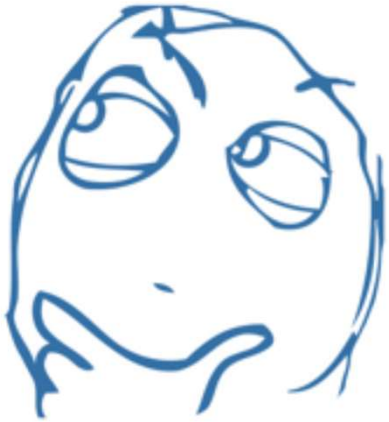
Affinità

- 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

Ok, ma quando convengono?



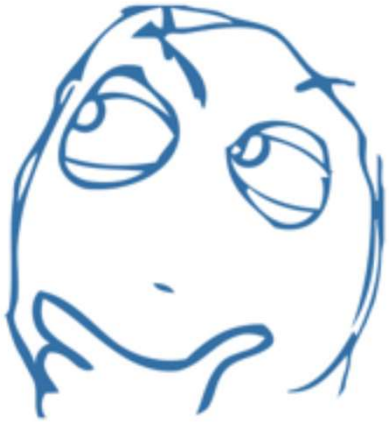
VANTAGGI DEL DESK-CHECK

- **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**

E i metodi formali?



METODI FORMALI

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

Ad esempio, il protocollo «two-phase locking»

- si dimostra corretto
- se istanziato correttamente garantisce assenza di malfunzionamenti dovuti alla race condition

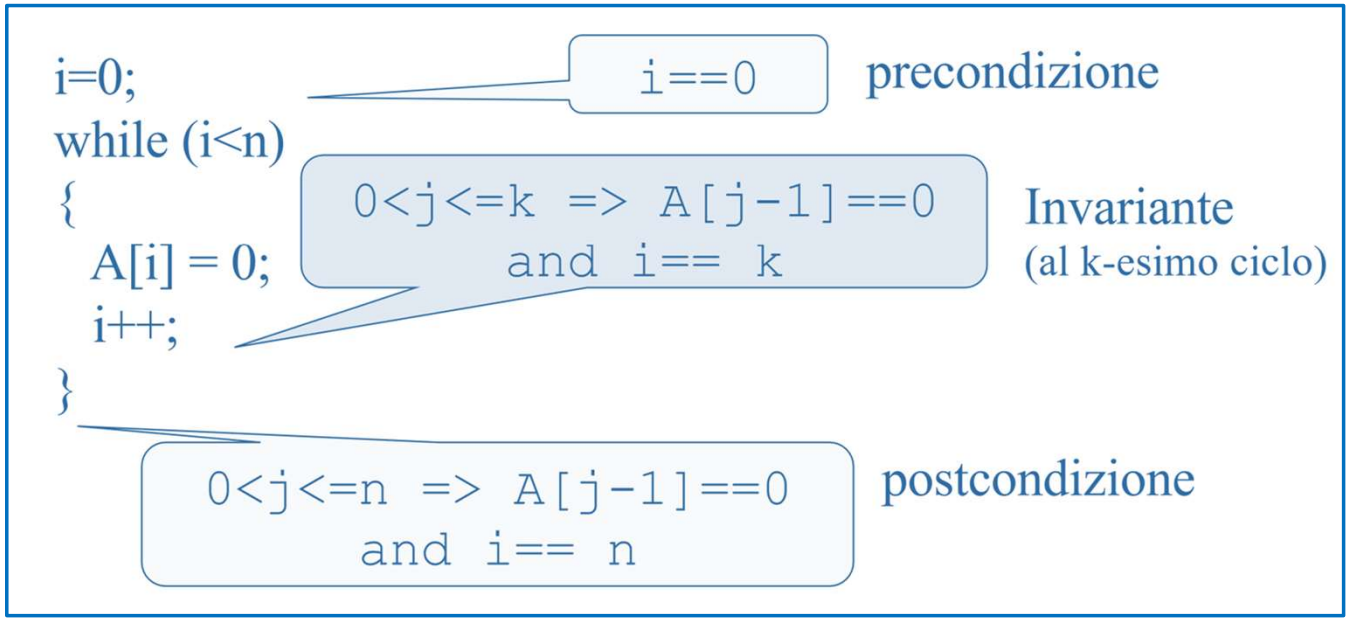
Allo stesso tempo

- ci sono applicazioni che non usano «two-phase locking» e sono comunque corrette
- occorre provare che il programma applica correttamente il protocollo
  → di solito è più facile che provare l'assenza di malfunzionamenti in generale

METODI FORMALI (CONT.)

Esempi di metodi formali:

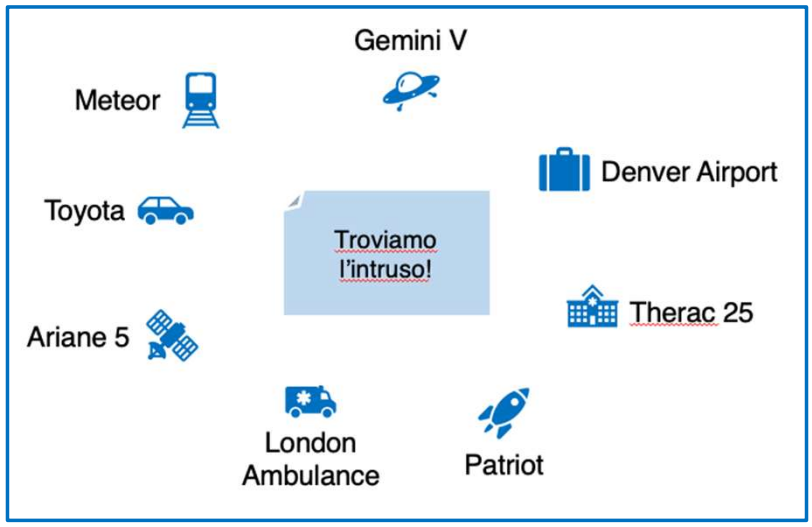
- Triple di Hoare



METODI FORMALI (CONT.)

Esempi di metodi formali:

- Triple di Hoare
- B method

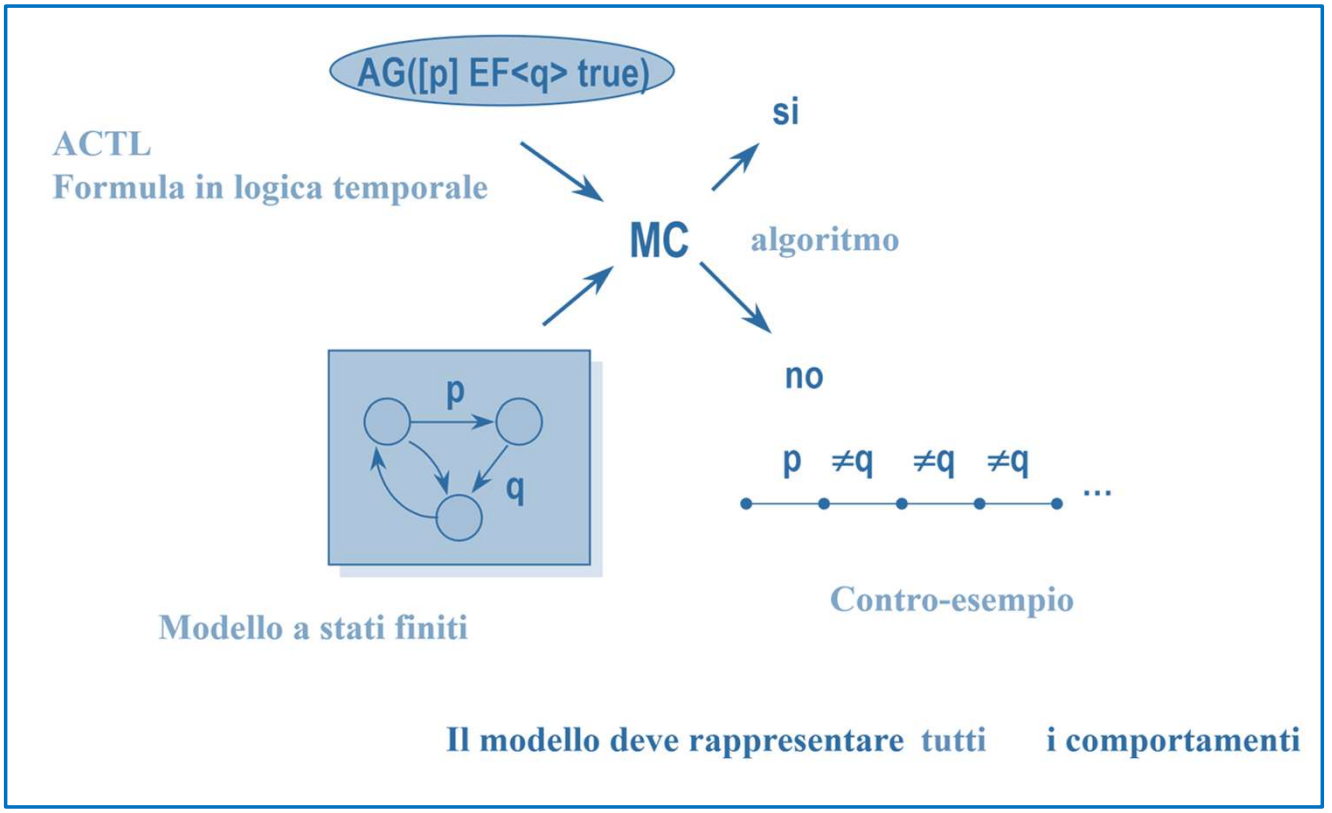


L'applicazione più nota del B method è la metropolitana automatica METEOR, linea 14 di Parigi

METODI FORMALI (CONT.)

Esempi di metodi formali:

- Triple di Hoare
- B method
- Model checking



Clarke/Emerson, Queille/Sifakis (1986)


Q
A

RIFERIMENTI

Contenuti

- **Capitolo 19** di "Software Engineering" (G. C. Kung, 2023)

Approfondimenti

- **Capitoli 1, 2 e 18** di "Software Testing and Analysis: Process, Principles and Techniques" (M. Pezzè, M. Young, 2008)