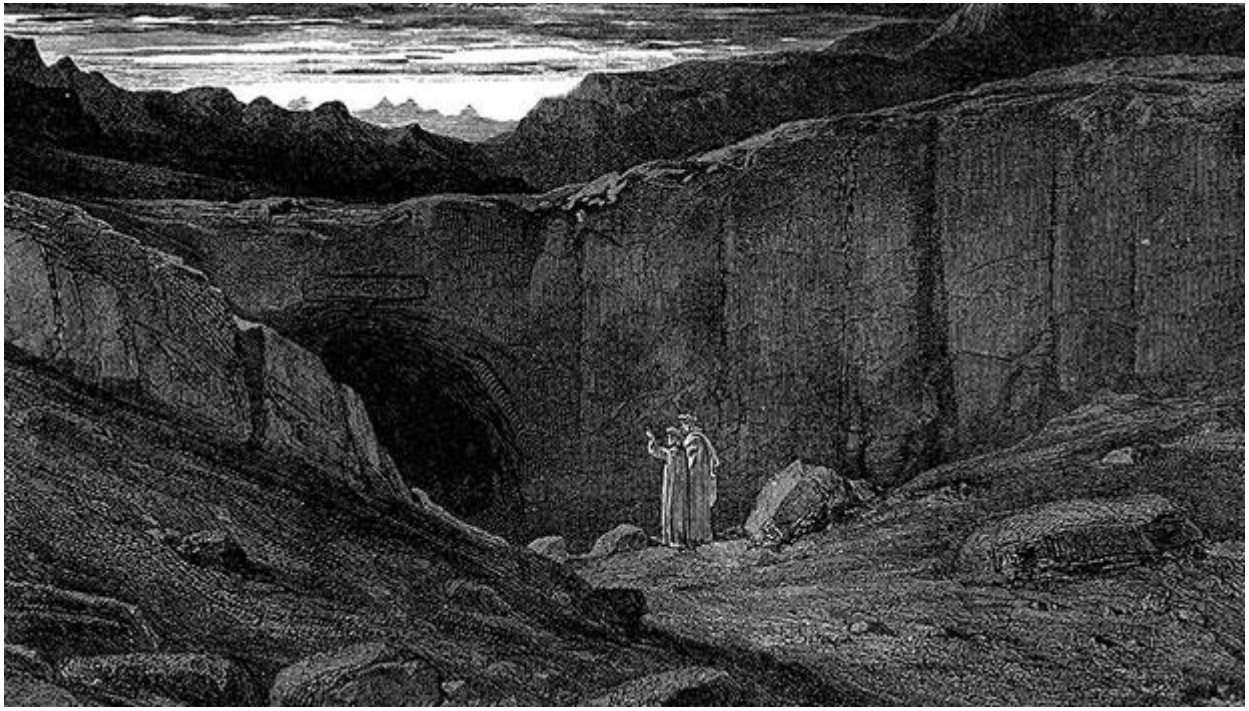
Introduzione alla fase e ai concetti di verifica e validazione

Roberta Gori, Laura Semini, thanks to Mauro Pezzè & Michal Young
Ingegneria del Software
Dipartimento di Informatica
Università di Pisa

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 I

- 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:

```
// halts() restituisce true se il suo input termina, false
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) solo se C(K,K) è falso, 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 II

- 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).

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

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

Indecidibilità

- 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
  - 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.

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 p. es. indecidibile:

- Dire se due programmi C producono lo stesso risultato in corrispondenza degli stessi dati di ingresso
  - Per esempio i due metodi visti

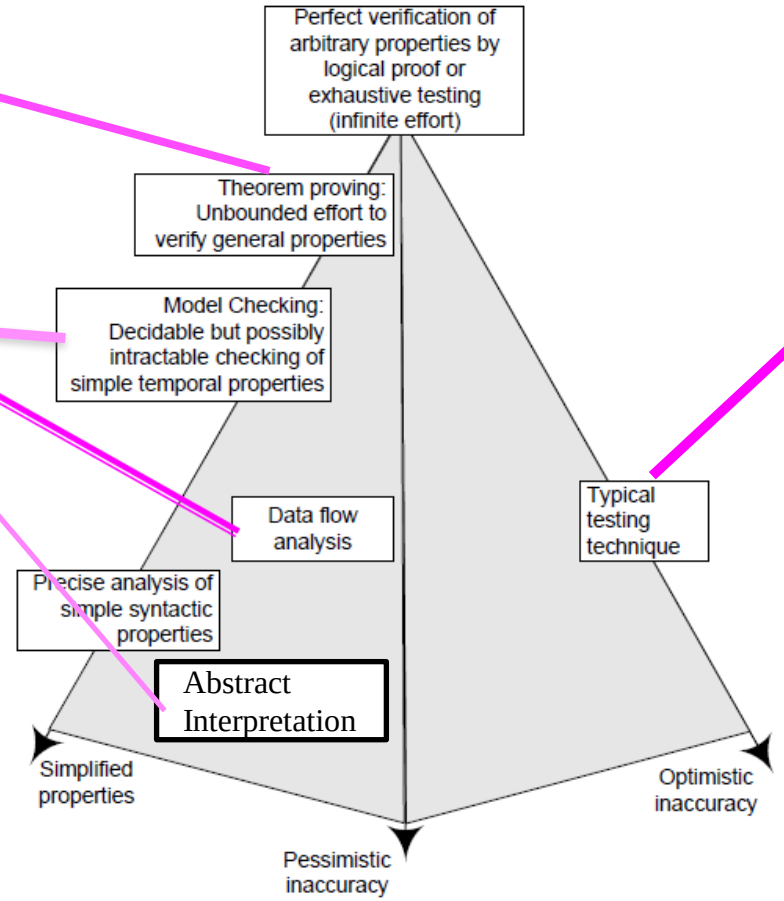
```
public void printHW(myHome Home){
        System.out.println("Hello, World");
  }

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

Ciò premesso

- Vediamo cosa possiamo fare

LPP: triple di Hoare

In corsi della magistrale



In questo corso

LPP e 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 $\models F$ oppure $\not\models F$?
    - 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 usati

- 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

Varietà di approcci: non ci sono ricette fisse

- 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

Non ci sono ricette fisse ma fatevi guidare da queste cinque domande

1. Quando iniziare verifica e convalida? Quando sono complete?
2. Quali tecniche applicare?
3. Come possiamo 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 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

1: 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

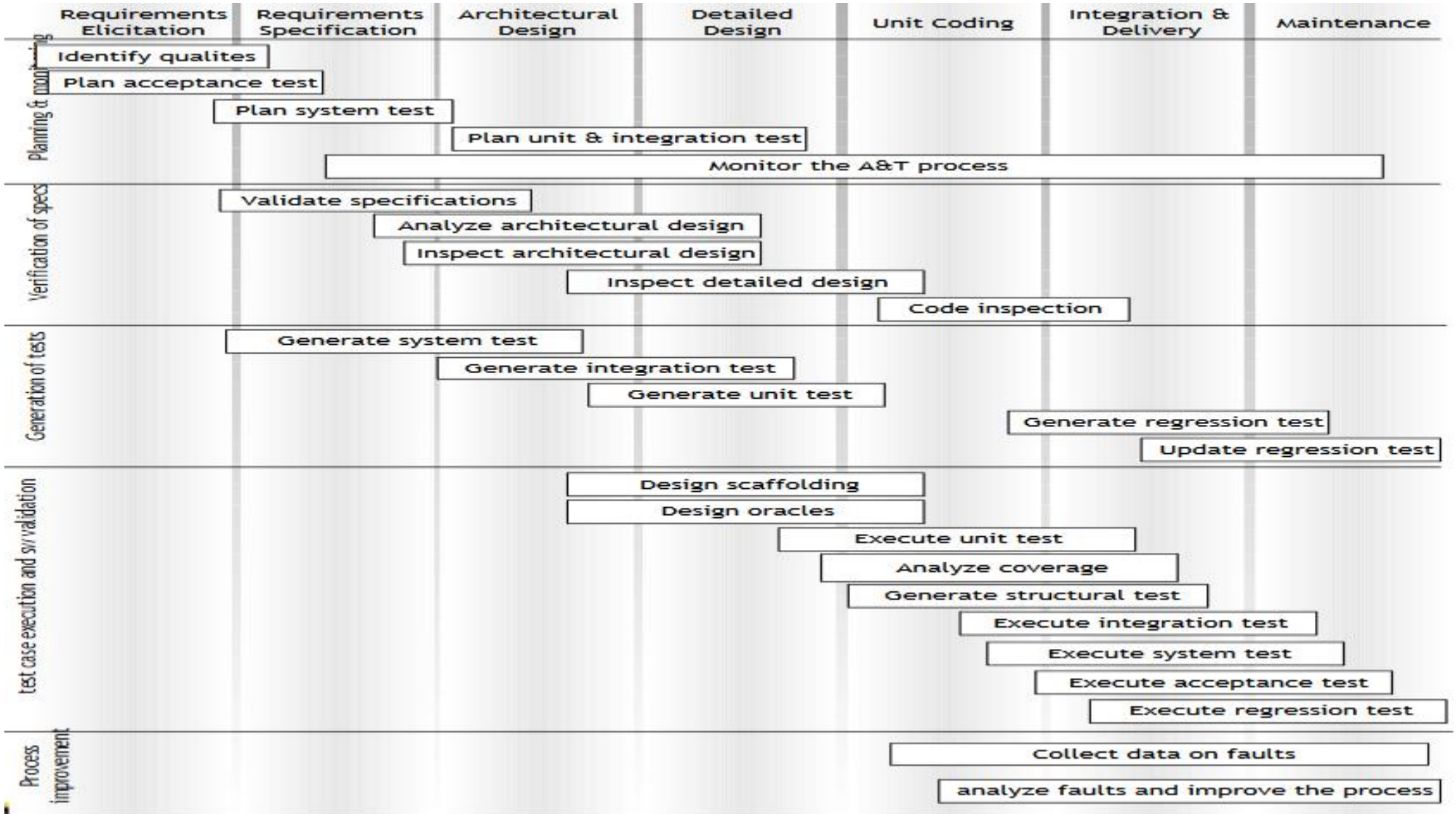
1: V&V dopo il rilascio

- 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. Differenze negli 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?
Le misure vanno ben definite (es. affidabilità)

- 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?

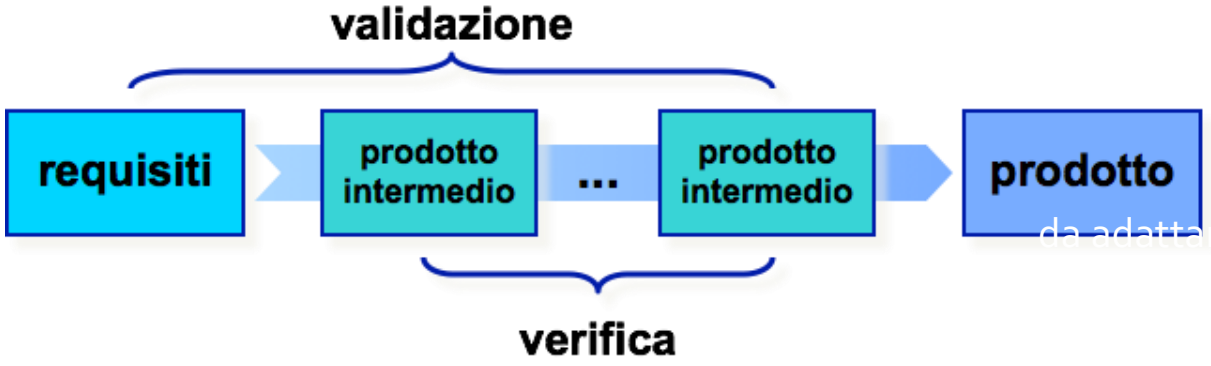
- Attività dopo la consegna
  - test e analisi del codice nuovo e modificato
  - riesecuzione dei test di sistema
  - memorizzazione di tutti i bug trovati
  - test di regressione
    - Quasi automatico
  - distinzione tra "major" e "minor" revisions
    - 2.0 vs 1.4
    - 1.5 vs 1.4

5. Come può essere migliorato il processo di sviluppo?

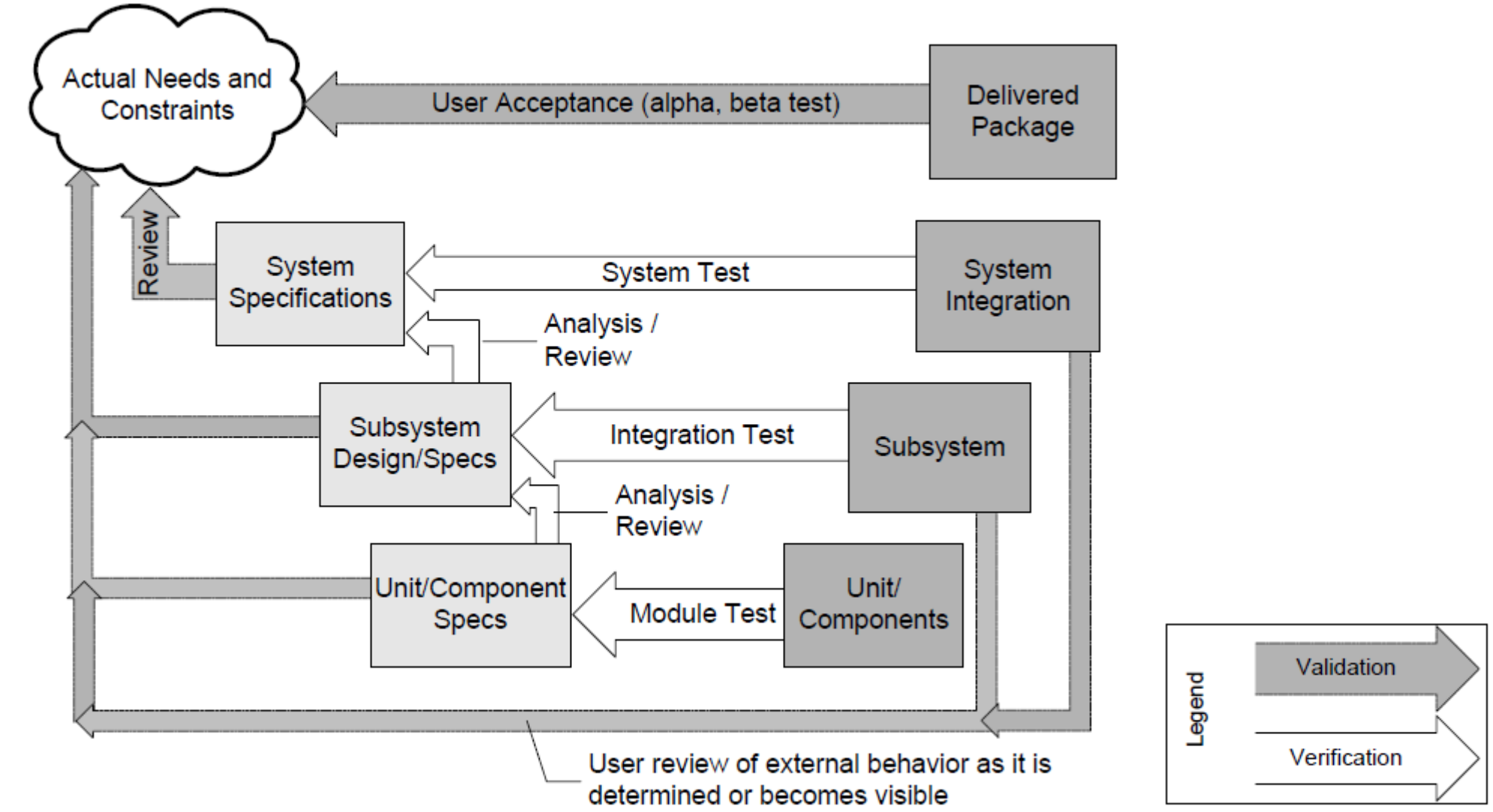
- Si incontrano gli stessi difetti progetto dopo progetto
  - identificare e rimuovere i punti deboli nel processo di sviluppo
    - Per esempio cattive pratiche di programmazione
  - identificare e rimuovere i punti deboli del test e dell'analisi che consentono loro di non essere individuati

Verifica vs convalida

- La **convalida** risponde alla domanda:
  - stiamo costruendo il sistema che serve all'utente
  - Validation, a volte tradotto con validazione
- La **verifica** risponde alla domanda
  - Stiamo costruendo un sistema che rispetta le specifiche?



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 **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 ;
        }
```

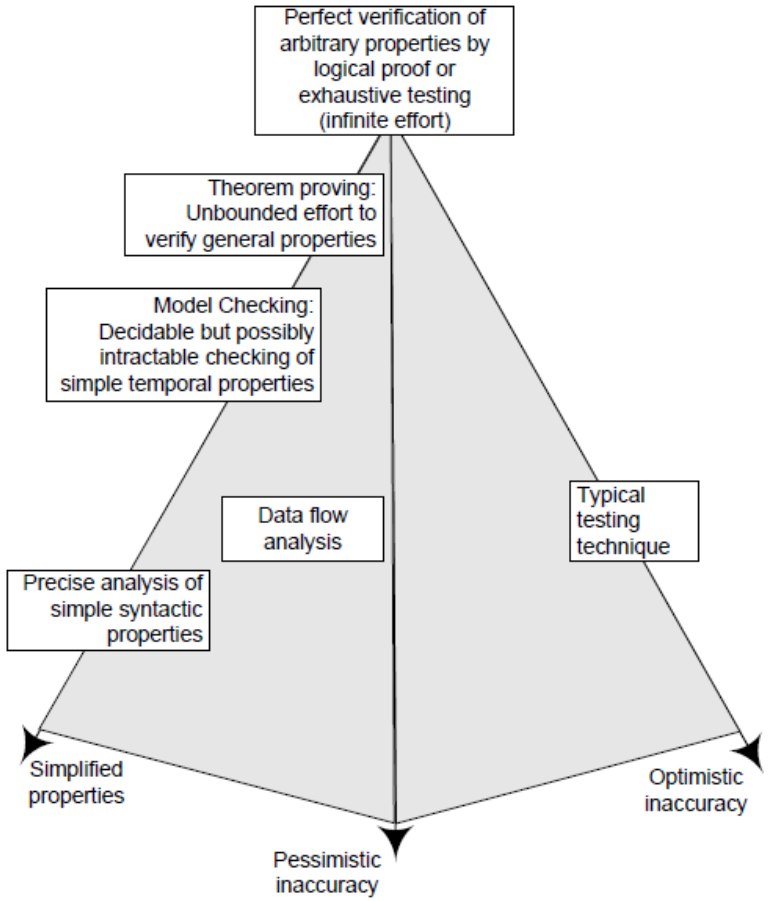
- Con input 3, restituisce 9
  - malfunzionamento del metodo raddoppia
- Il malfunzionamento è causato dalla presenza di un difetto:
  - In questo caso l'operatore * invece di +

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!)

Limiti del testing

- Cosa significa "optimistic inaccuracy"



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

$$10^{21} \times 10^{-9} = 10^{12} \approx 30.000 \text{ anni}$$

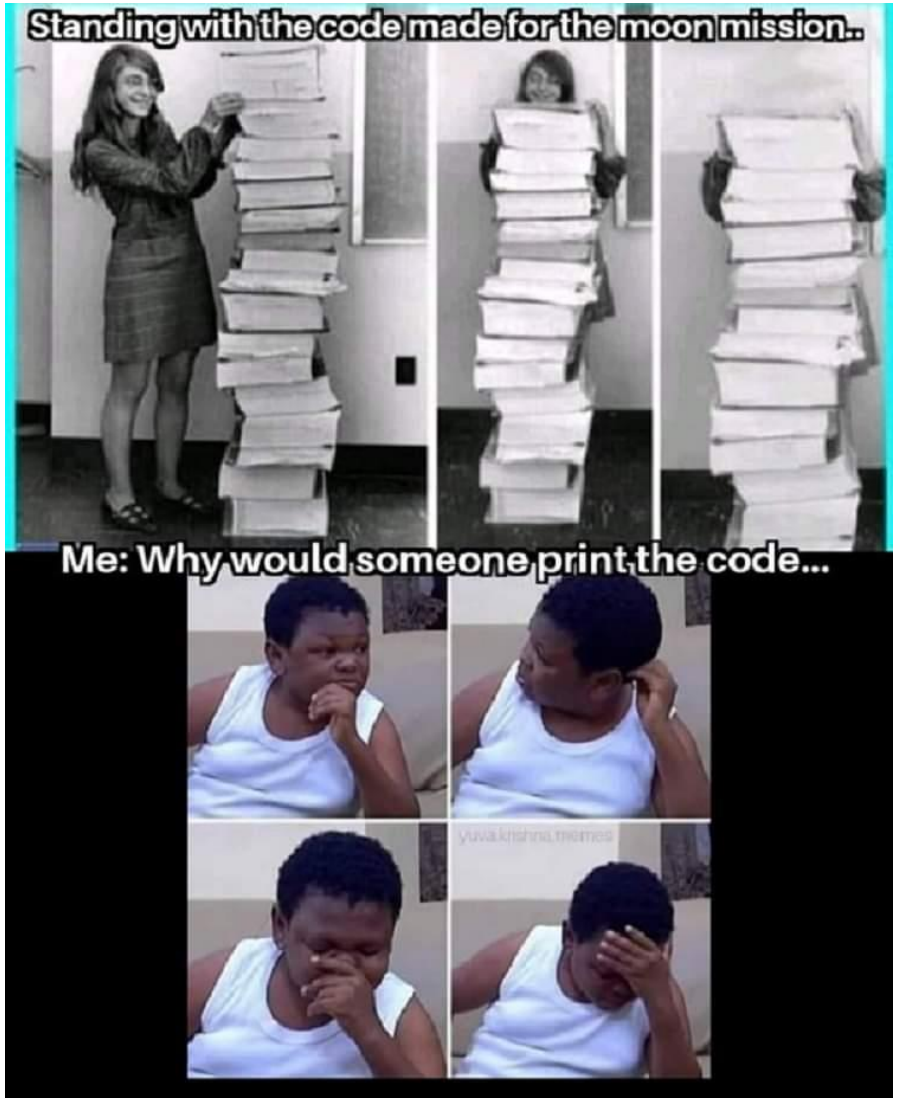
Tesi di Dijkstra

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

La 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



Lo sbarco sulla luna è avvenuto nel 1969. I pc non erano comuni fino agli anni '80. E i terminali negli anni '70.

I terminali potevano mostrare solo 24 righe di 80 caratteri ciascuna fino a molto più tardi.

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.

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

- Obiettivi
  - rivelare la presenza di difetti
  - eseguire una lettura mirata del codice (guidata da una lista di controllo che va aggiornata via via)
- Strategia
  - **focalizzare la ricerca su aspetti ben definiti (error guessing)**
    - Ex: off-by-one error (aka Obi-Wan error)
- Agenti
  - verificatori diversi dai programmatori

Attività dell'inspection

- 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

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			

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

Attività di 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

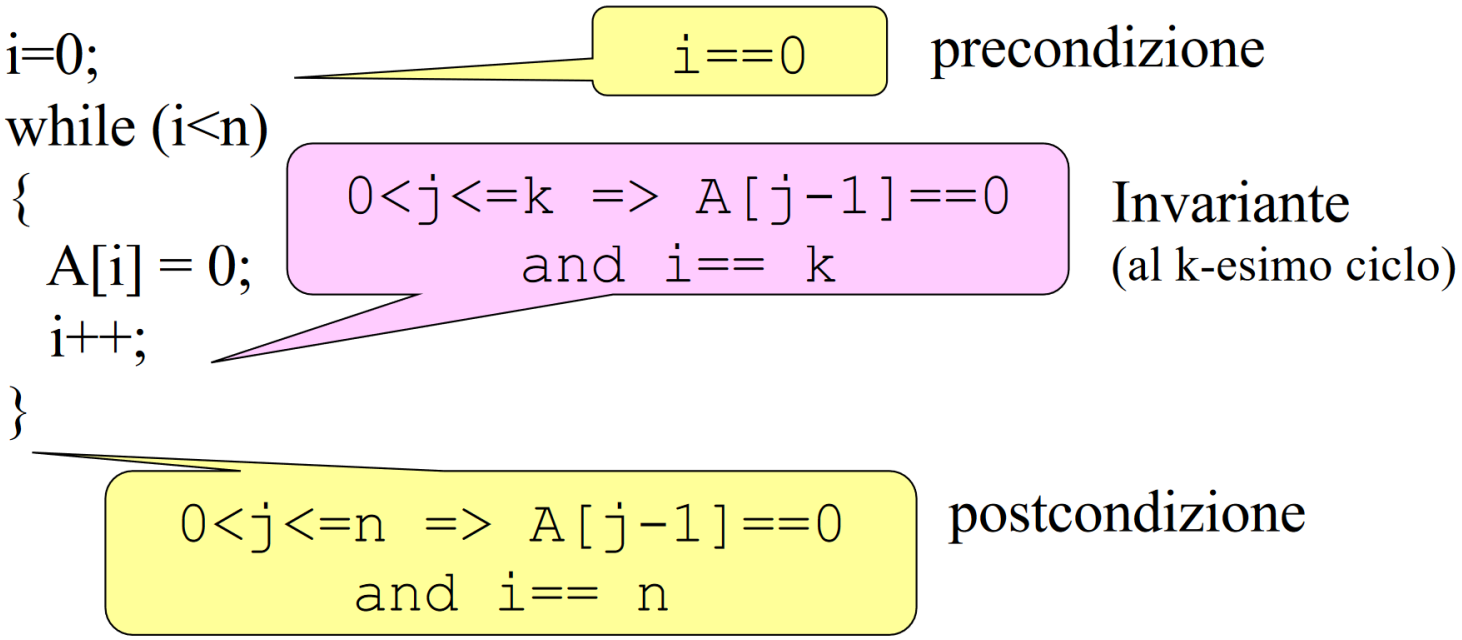
- Tecnica basata sulla dimostazione formale di correttezza di un modello finito (dimostrazione possibile) e istanziazione del modello.

- Esempio protocollo «two-phase locking»:
  - si dimostra corretto
  - se istanziato correttamente garantisce assenza di malfunzionamenti dovuti alla race condition

Metodi Formali

- Osservazioni (es: two-phase locking):
  - Ci sono applicazioni che non usano «two-phase locking» e sono corrette (approcci pessimistici)
  - 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



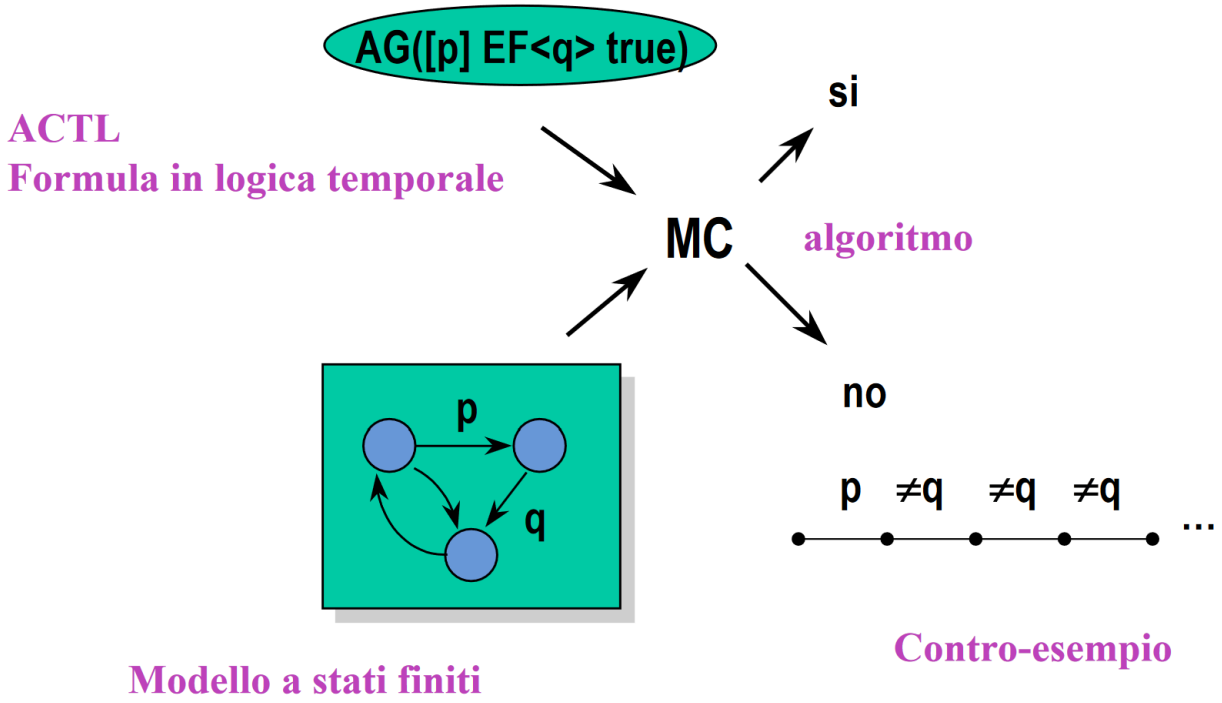
Metodi formali: B method

Chi è l'intruso?

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

Syllabus

- Cap 1-2 -18
  - Software Testing and Analysis: Process, Principles and Techniques-
  - Mauro Pezzè e Michal Young