

Modulo1: Introduzione al Testing

Lezione 1.1 – Il ruolo del testing

Angelo Gargantini

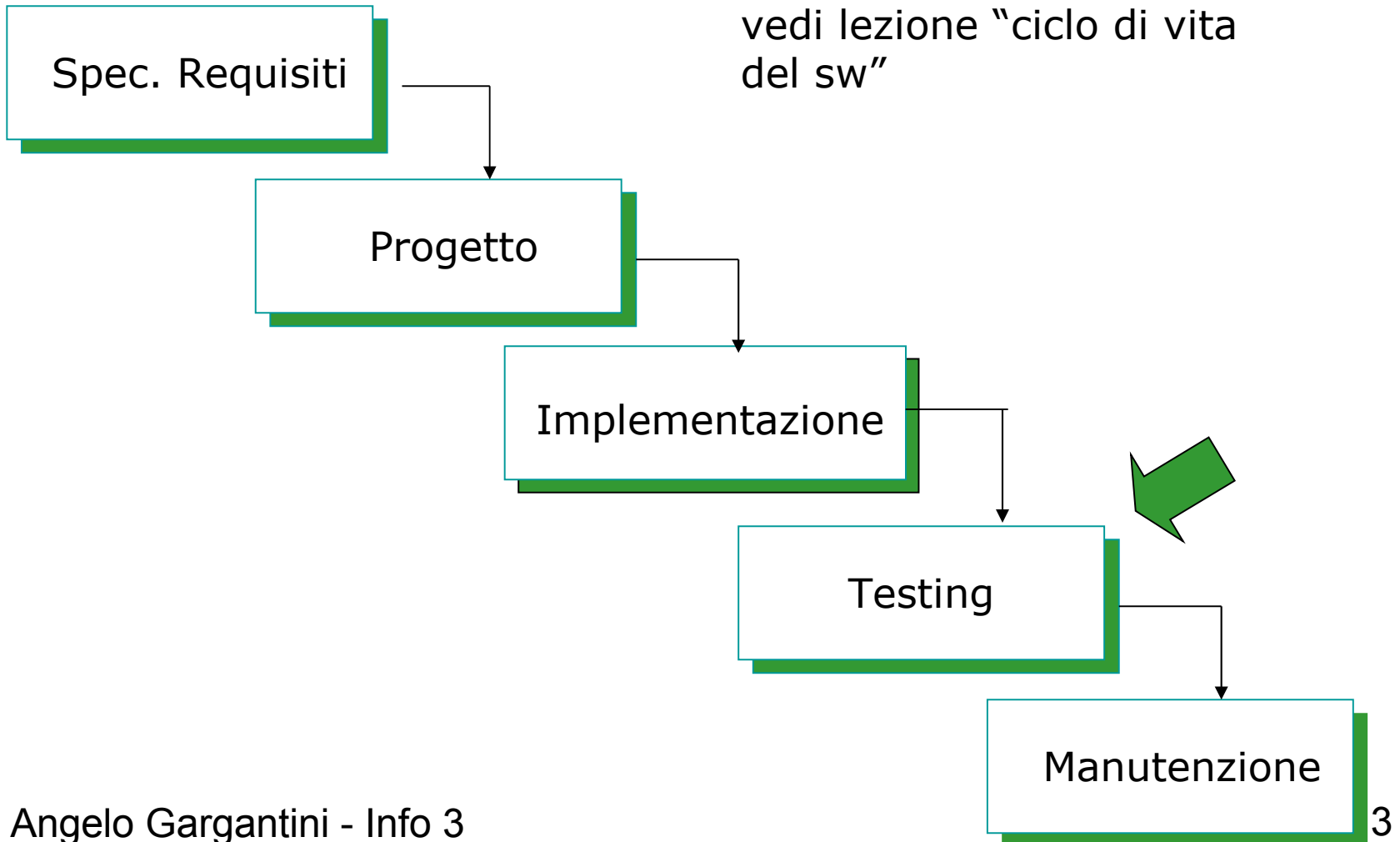
Testing in breve

Il **test di un programma o di un sistema consiste (in breve) nell'eseguirlo con alcuni casi di test e controllare che il comportamento sia corretto**

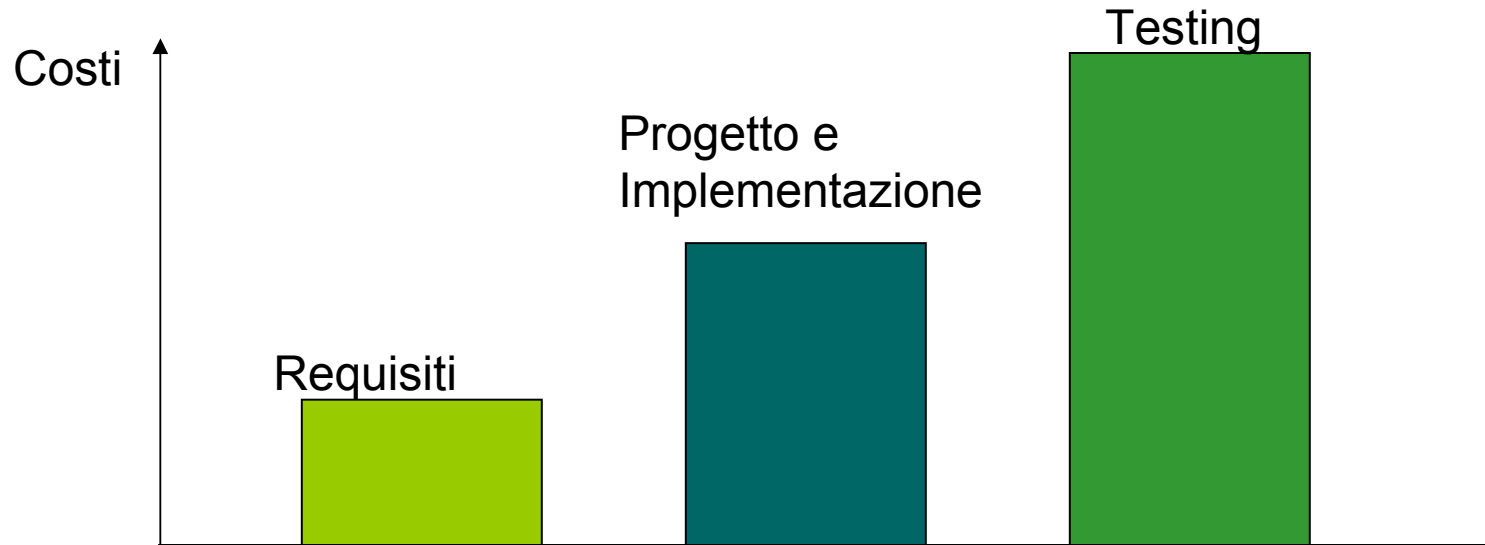
In italiano si dice anche **collaudo del software**

Ciclo di vita del Software

Ciclo di vita a cascata –
vedi lezione “ciclo di vita
del sw”



Costi dello sviluppo del Software



I costi del testing sono normalmente molto maggiori che delle altre fasi

Possibili concause sono:

- risparmio sulle fasi iniziali

- **frettolosità nel rilascio del software** (time to market)

Alcune domande di base

E' possibile avere software **senza difetti**?

Fino a che punto possiamo fidarci del testing o di altre tecniche di verifica?

- Ogni parte andrebbe verificata (anche la verifica)

Tutti i sw devono essere corretti al 100%?

- sistemi critici
- sistemi per la produttività individuale

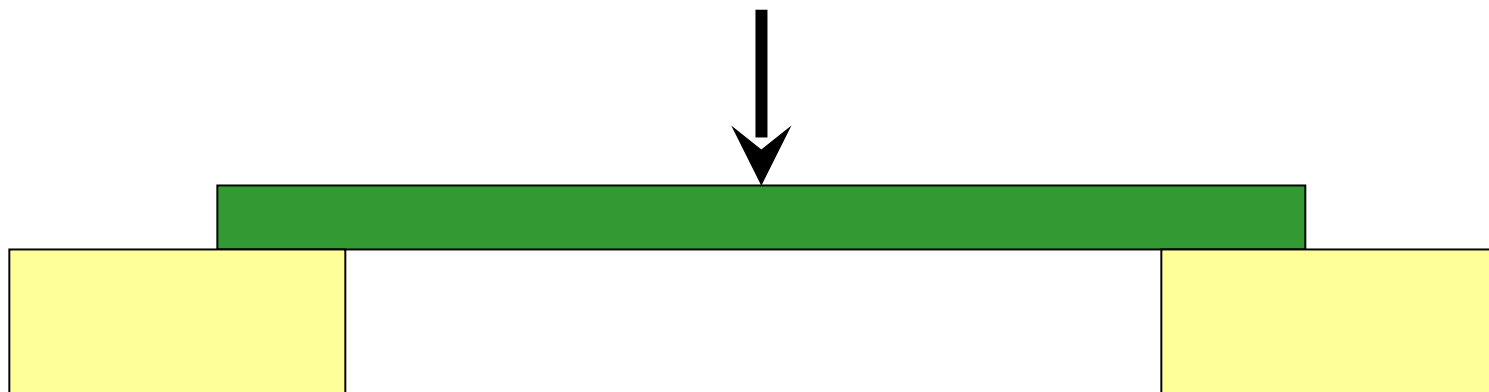
"software industry, the: unique industry where selling substandard goods is legal and you can charge extra for fixing the problems."

Test in Ingegneria

Il testing è una pratica diffusa in tutti i campi dell'ingegneria

Però in ingegneria classica gli oggetti hanno un comportamento "continuo":

- un ponte si testa in un punto particolare e se funziona lì funziona tutt'attorno (o addirittura tutto)



Test nell'ingegneria del software

1. Il software ha un comportamento molto **discontinuo** quindi la selezione dei "punti" in cui effettuare il test e' molto critica

Esempio

$$a := x / (x + 20)$$

Quale valore prendo di x per vedere se funziona?

Problema dei requisiti:

2. un ponte basta che resista, un software basta che non smetta di funzionare o richiedo qualcosa di più?

Limiti del testing

**Il testing è efficace a trovare bug
ma è inadeguato a provare l'assenza di bug**

- Posso eseguire il programma 1000 volte ed essere (s)fortunato e non scoprire mai un certo bug

**Non può sostituire una buona pratica di
progettazione e implementazione**

Approfondiremo questo argomento nell'unità 5.2

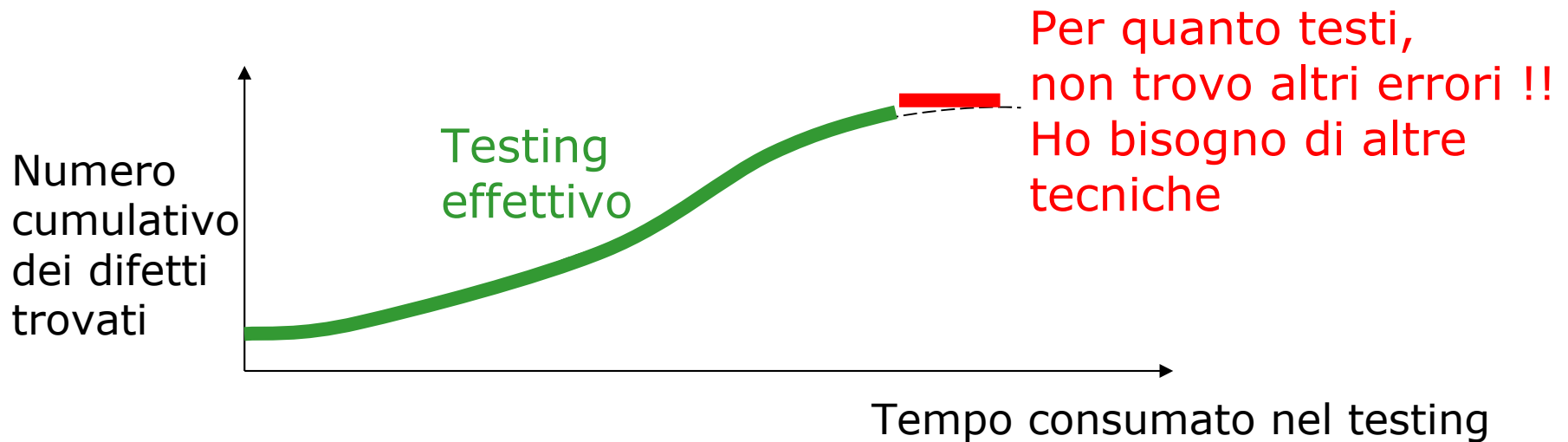
Alcune linee guida generali

Il testing dovrebbe:

- essere **automatizzato**
 - Uso di strumenti automatici che assistano i programmatori nella scrittura di test, nell'esecuzione, nella raccolta dei dati di output e nell'analisi della loro correttezza
- riguardare **ogni fase di sviluppo**
 - Non solo sul codice finale, ma anche sui requisiti, sui prototipi, ... → prossima lezione
- essere **esteso a tutti** i componenti di un sistema
- essere pianificato (**test plan**)
- seguire **standard** e metodologie dove possibile

Economia del Testing (1)

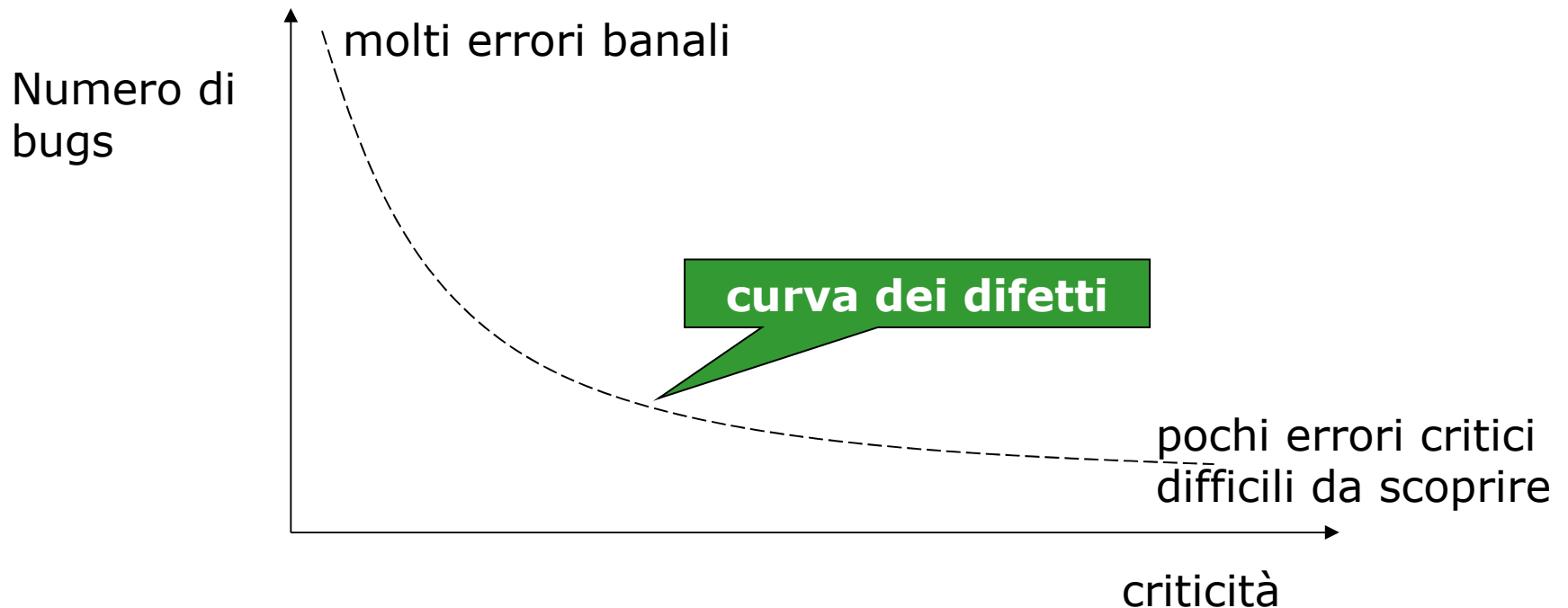
Curva di rimozione degli errori



Curva dei difetti

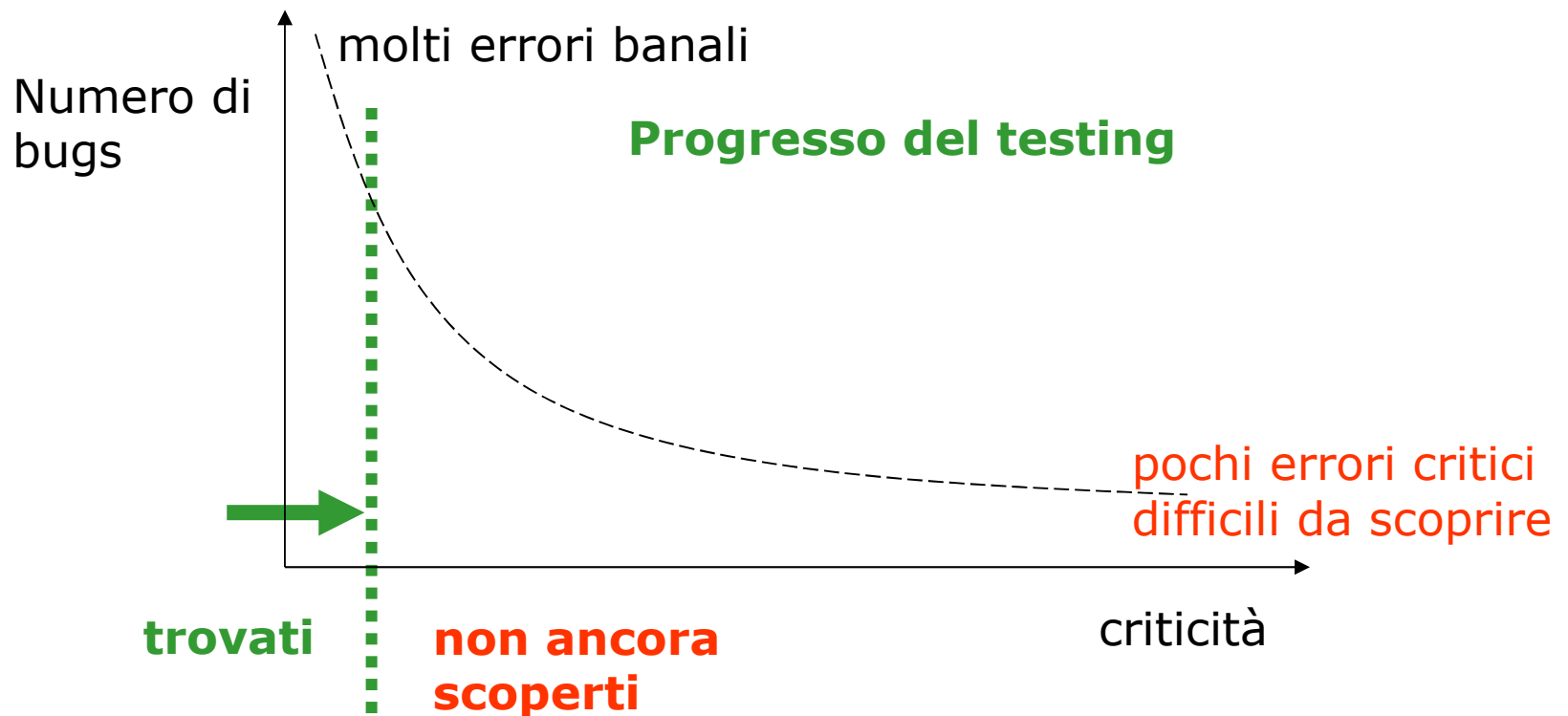
Curva dei difetti

ipotesi: i difetti più critici sono di meno e più difficili da scoprire



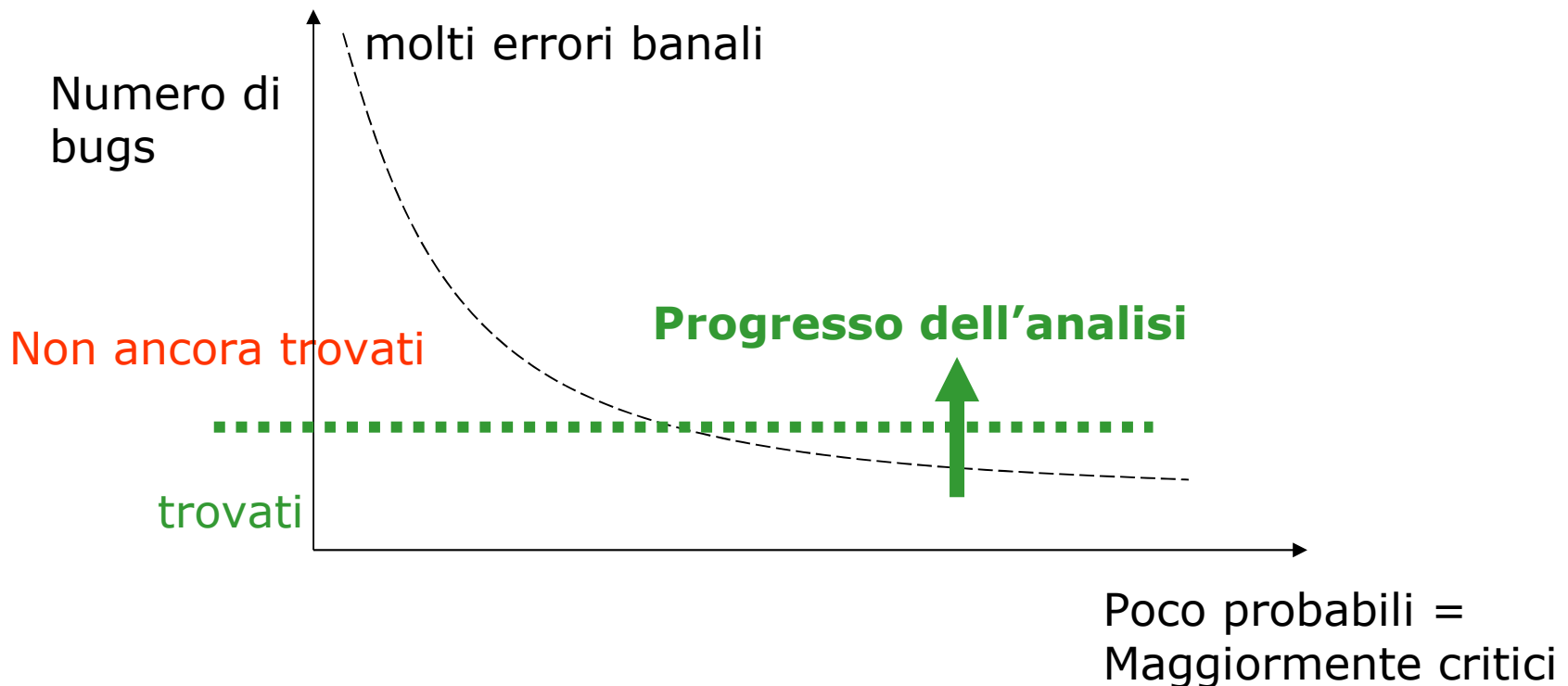
Economia del testing (2)

Il testing tende a scoprire i difetti con probabilità proporzionale alla loro densità:



Economia del testing (3)

Tecniche (più costose) di **analisi statica** invece cercano di scoprire difetti con probabilità proporzionale alla loro criticità



Economia del testing (4)

Attenti però:

non sempre si ha questa distribuzione

- Potrei avere difetti difficili da scoprire ma non critici

Fattori da considerare:

1. quanto costa il testing?

2. quanto costa avere un sw con un difetto?

- costo del danno che provoca x
probabilità che accada
- quindi è "inutile" spendere troppo tempo a cercare difetti che non provocano danno
 - Per questo motivo tecniche più sicure non vengono impiegate e il **sw non critico** viene rilasciato con ancora molti difetti

In sintesi

- Abbiamo visto che il testing:
 - consiste nel collaudare il sw mediante prova
 - si applica normalmente alla fine del ciclo di vita del sw
 - ha costi considerevoli
- Ricordate che il testing del sw:
 - è molto sensibile ai punti di testing
 - il successo di una prova non è molto significativo
 - è utile per trovare errori ma difficilmente garantisce l'assenza
 - alcuni errori più critici potrebbero essere molto difficili da trovare mediante testing
 - dovrebbe essere applicato durante tutto il ciclo di vita



T1.2 – Definizioni Base

Angelo Gargantini

Terminologia base del testing

Dobbiamo concordare alcune parole che useremo da qui in avanti:

- alcuni termini hanno un significato chiaro:
 - testing, debugging, ...
- altri sono definiti in modo non ambiguo da standard
 - Ad esempio dalla IEEE Standard Glossary of Software Engineering Terminology
 - difetto, bug, ...
- altri sono usati in modo ambiguo
 - Proponiamo un uso e cerchiamo di essere consistenti
 - errore, ...

Malfunzionamento

Un **failure** o **guasto** o **malfunzionamento** è il funzionamento non corretto del programma

- legato quindi al comportamento che si osserva durante l'esecuzione

Esempio:

```
// programma che dovrebbe restituire il doppio
// del valore passato come parametro
int raddoppia(int x) {
    return x*x ;
}
```

se chiamo raddoppia(3) noto un **malfunzionamento**
se chiamo raddoppia(2) **non vi e' malfunzionamento**

Difetto

Un **difetto** o **anomalia** o **fault** o **bug** è un elemento del programma sorgente non corrispondente alle aspettative

- riguarda quindi la parte statica.
- **uno o più difetti possono causare malfunzionamenti**
- Nell'esempio:

```
int raddoppia(int x) {  
    return x*x ;  
}
```

il difetto e' *x invece che *2.

Nota

- un programma può avere molti difetti e non presentare alcun malfunzionamento.
- scopo del testing e' quello di evidenziare difetti mediante malfunzionamenti.

Errore: fattore (umano) che causa una deviazione tra il software prodotto e il programma ideale (uno o più errori possono produrre uno o più difetti nel codice)

- **Esempio:** errore di analisi dei requisiti, progetto, battitura,...
- Quando il programmatore commette un errore, il programma ha un difetto che può generare un malfunzionamento

Testing e debugging

Testing di programmi:

- eseguire il programma con dei casi di test e analizzare i risultati per trovare i difetti (bug)

Debugging:

- correggere i difetti ed eventualmente scoprire gli errori

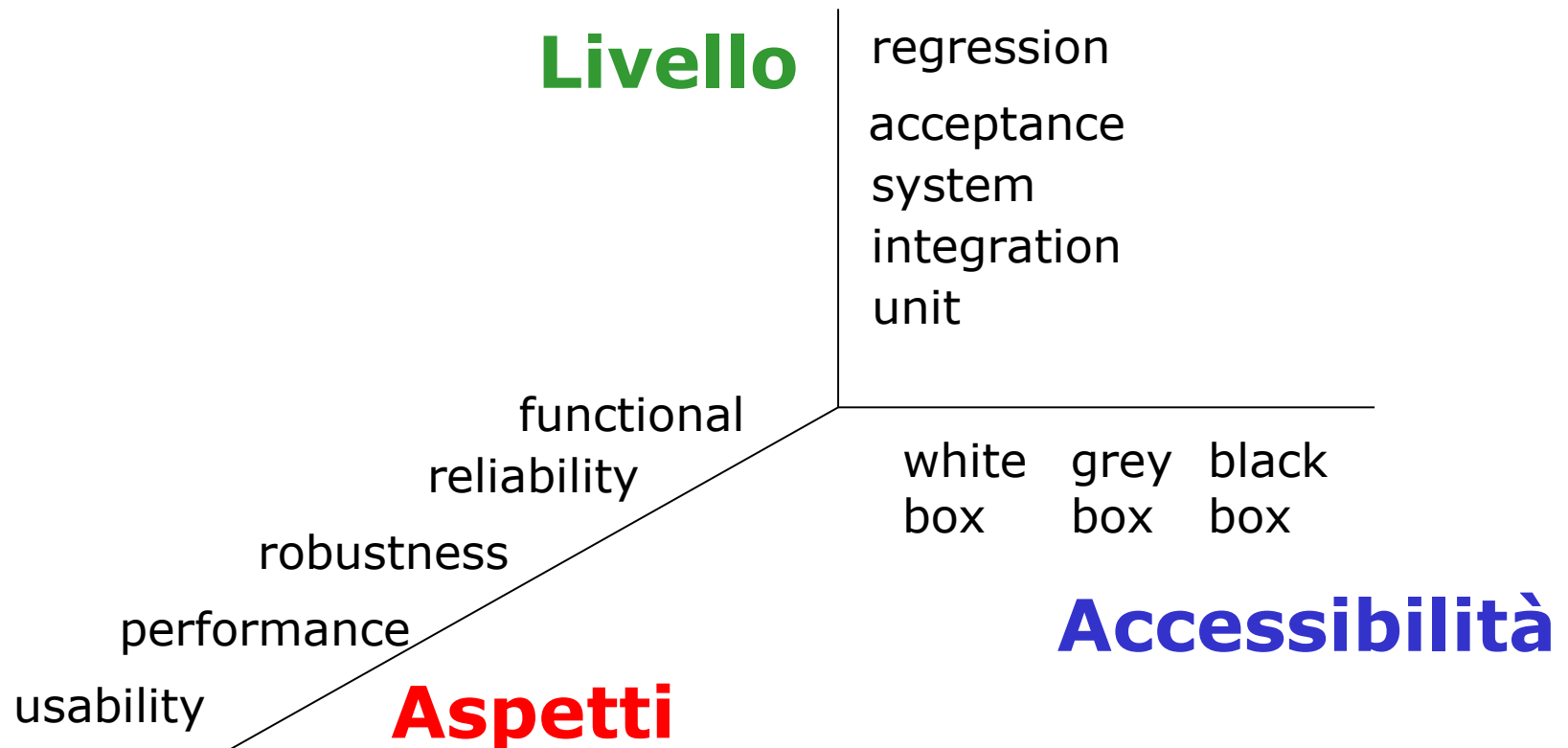
Scopi del testing

Testing di programmi ha due scopi principali

- mettere in evidenza i difetti (bug) mediante i malfunzionamenti, per scoprire eventuali errori:
 - cercare quei comportamenti che mettono in evidenza difetti
- poter valutare l'affidabilità di un sw (reliability) e fornire confidenza (test di accettazione)
 - dell'affidabilità del prodotto e della (probabile) correttezza
 - del aver rilevato l'assenza di particolari tipi di malfunzionamenti
 - cercare quei comportamenti più critici o più frequenti per controllare che causino o meno malfunzionamenti

Tipi di testing

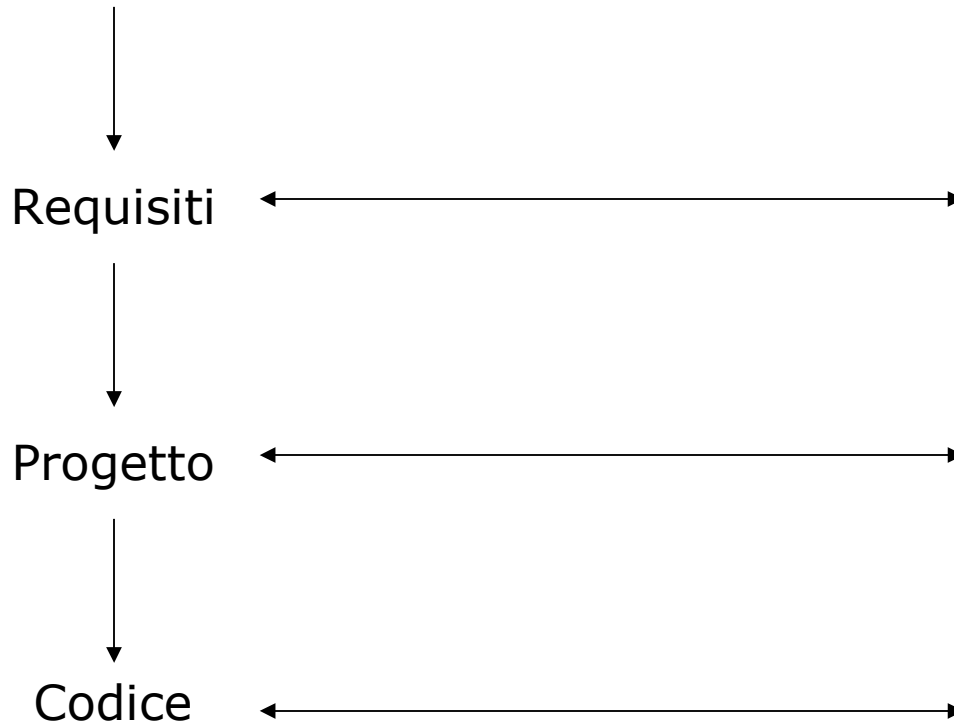
Esistono tanti tipi di testing a seconda del **livello** a cui si effettuano, del tipo di **accesso** al sistema da testare e degli **aspetti** che si vogliono testare



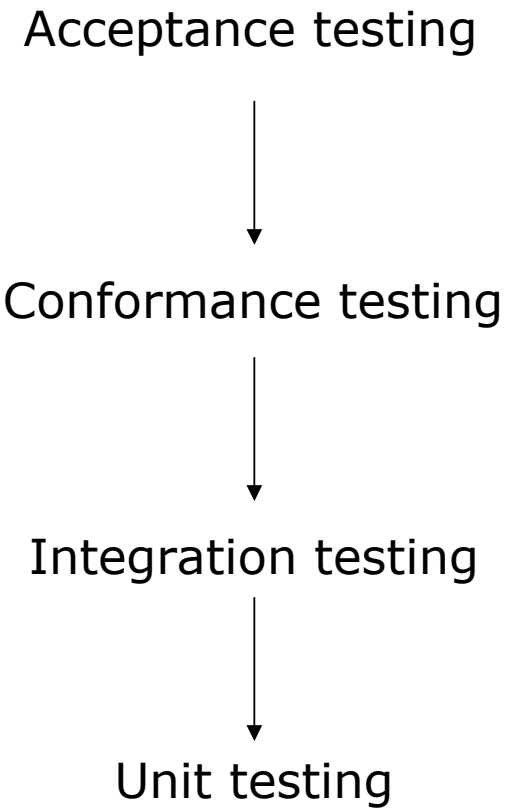
I livelli del Testing

livello da testare

Ciò di cui l'utente
ha bisogno



Tipo di test



Livelli di granularità

Test di

accettazione: il comportamento del software è confrontato con i requisiti dell'utente finale

conformità: il comportamento del software (tutto) è confrontato con le specifiche dei requisiti

sistema: controlla il comportamento dell'intero sistema (hw + sw) come monolitico.

integrazione: controllo sul modo di cooperazione delle unità (come previsto dal progetto)

unità: test del comportamento delle singole unità

regressione: test del comportamento di release successive

Testa il codice a livello di singola unità

In OO (come JAVA)

- la singola unità è una class
- UT testa quindi i singoli metodi delle classi

Per ogni metodo testato - detto **test unit** - si introducono

- **test driver**: metodo che chiama il test unit con opportuni parametri
- **test stub**: (opzionale) metodo che sostituisce eventuali metodi usati dal test unit per testare in modo isolato e controllato
 - La scrittura di stub può essere onerosa e viene spesso evitata

Unit Testing - Esempio

Esempio

```
foo(int x2, int y2) {  
    .....  
    gig(x2+2);  
    .....  
}  
testFoo() {  
    .....  
    foo(x1+1, y1-1);  
    // controllo  
}  
gig(int x3) {  
    .....  
}
```

→ **foo: test unit**

Metodo da testare

→ **testFoo: test driver**

Metodo che testa foo

Simula una unità chiamante

→ **gig: test stub** (opzionale)

Simula un metodo chiamato da foo in modo di isolare il caso di test dal resto del sistema

Test d'integrazione (1)

Testa l'integrazione tra le diverse unità

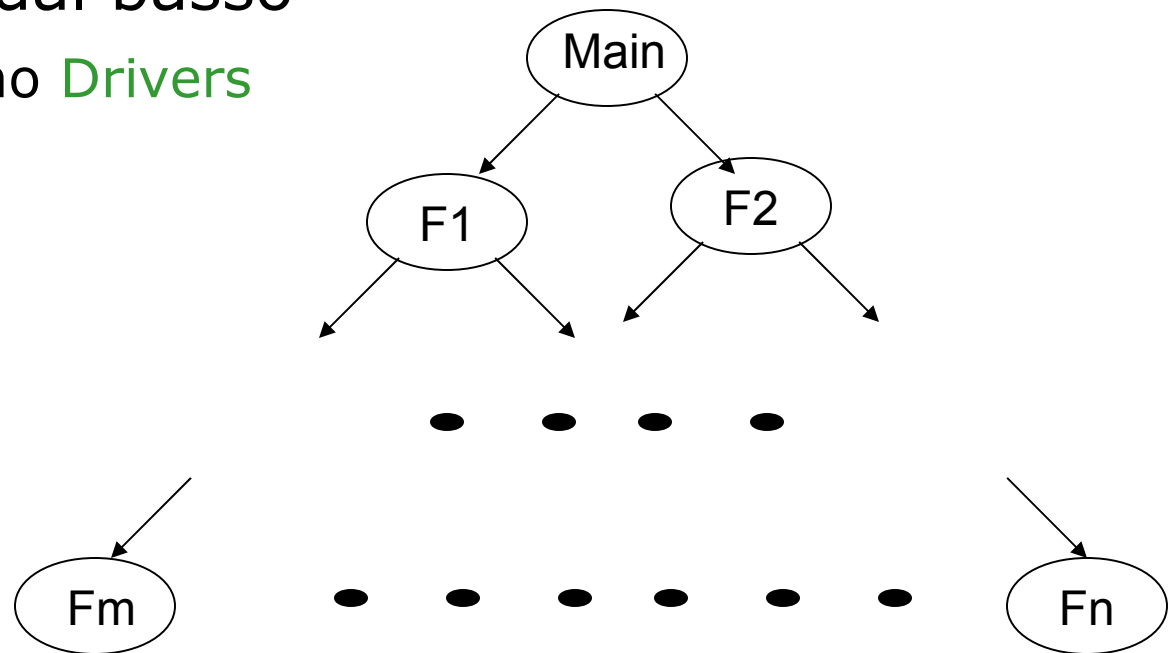
Esempio:

- compatibilità dei tipi importati/esportati
 - spesso eseguita dal linker o dal compilatore
- difetti nei domini di import/export
 - Esempio:
 - F1 chiama F2 con un parametro intero
 - F2 assume che il parametro sia >0 , mentre F1 chiama con parametro = 0
- rappresentazione di dati importati/esportati
 - Esempio: F1 chiama F2 e con un parametro Elapsed_time
 - F1 pensa siano secondi, F2 invece millisecondi
 - Successo veramente su sonda Pioneer????!!!!

Test d'integrazione (2)

Alcune strategia per il test di integrazione

- **top-down**: inizia dall'alto
 - Si devono sviluppare **Stubs**
- **bottom-up**: dal basso
 - Si sviluppano **Drivers**
- **big-bang**



Regression Testing

Sviluppo vs Manutenzione

- costi di sviluppo: 1/3
- costi di manutenzione: 2/3



Testing durante la manutenzione

- obiettivi nel testare programmi **modificati**:
 - accertarsi di aver eliminato i difetti segnalati
 - essere sicuri di non introdurre nuovi difetti
 - evitare di dover buttar via i vecchi casi di test e riscriverne nuovi da capo
 - riusare i vecchi casi di test e confrontare che il comportamento non sia cambiato (eccetto malfunzionamenti)
 - evitare di dover ritestare tutti i programmi anche quelli non modificati

In sintesi

- Abbiamo visto le definizioni di:
 - difetto o bug: elemento sintattico nel sorgente sbagliato;
 - malfunzionamento: comportamento sbagliato
 - errore e debugging
- Visto diversi tipi di testing a seconda del livello:
 - di accettazione, di conformità, di sistema
 - di integrazione
 - di unità
 - di regressione

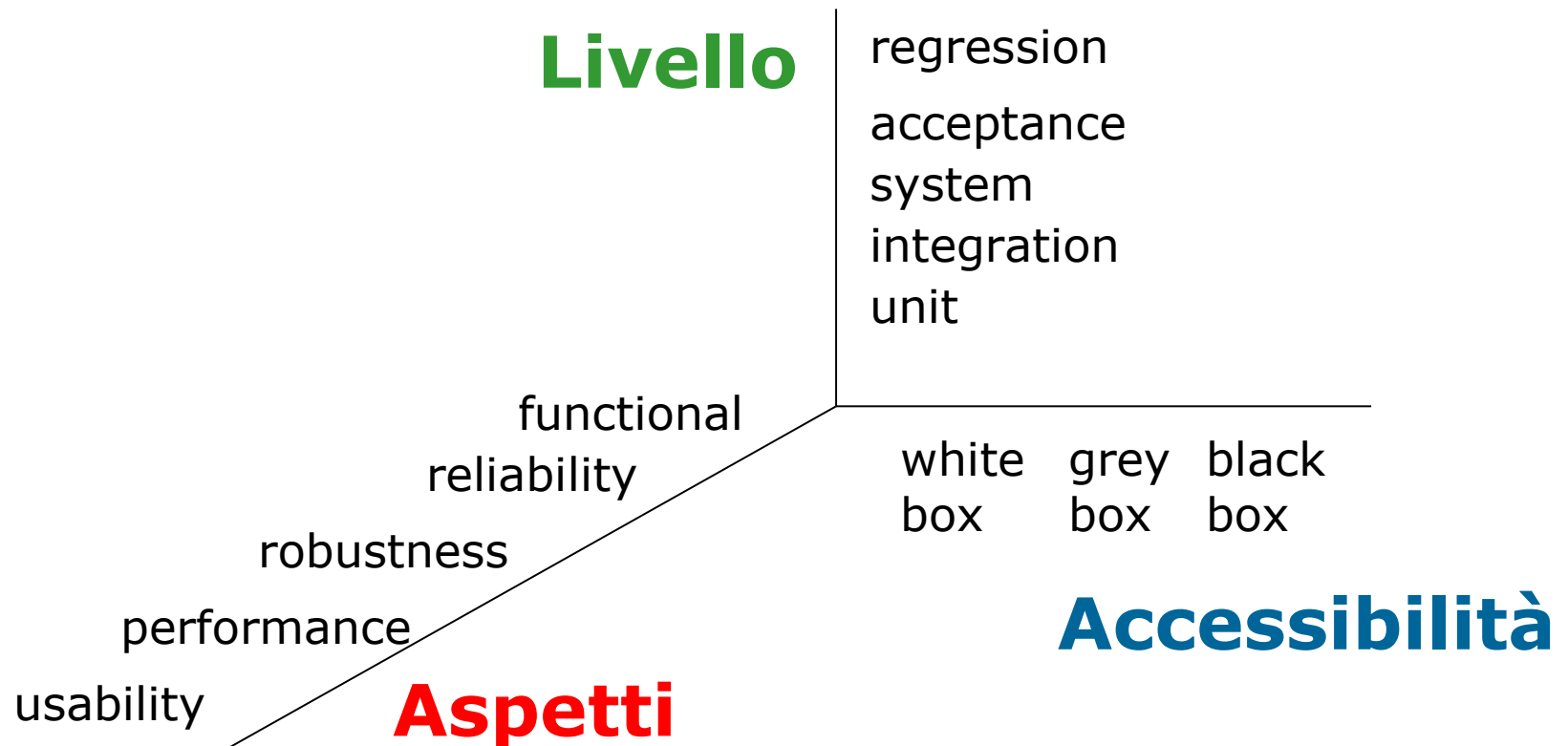


TIPI DI TESTING

Angelo Gargantini

Tipi di testing

Esistono tanti tipi di testing a seconda del **livello** a cui si effettuano, del tipo di **accesso** al sistema da testare e degli **aspetti** che si vogliono testare



Aspetti da testare

Funzionalità

Affidabilità – reliability

Robustezza

Performance

Usabilità

Vedi unità “Proprietà del sw sicuro”

Accessibilità del Testing

A seconda del tipo di “accesso” all’unità testata si ha:

white box testing, o structural testing, o program-based testing

- si assume che il programma sorgente sia disponibile

black box testing, o functional testing, o specification-based testing

- si assume che non si guardi il programma sorgente ma solo quello che dovrebbe fare

grey box testing

- un mix tra i due

White box testing

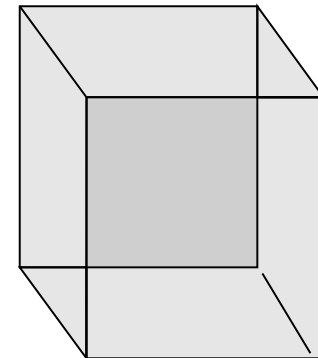
E' basato sulla struttura interna del programma

- deriva i casi di test dal programma
- controlla e osserva i programmi durante l'esecuzione

In genere analizza "quanto programma è stato eseguito" o coperto

- non garantisce che il programma faccia quello effettivamente richiesto
- ci sono diverse alternative che vedremo

White Box



Black-Box testing

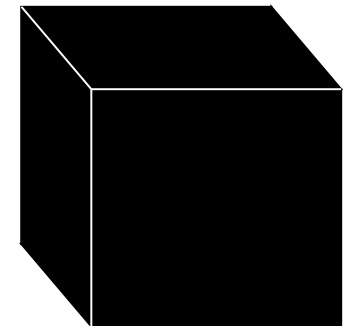
Ignora la struttura del programma e considera solamente i suoi requisiti

- deriva i casi di test dai requisiti
- controlla e osserva il programma solo attraverso la sua interfaccia esterna (input/output)

In genere misura quanti input/output sono stati utilizzati

- non tutti gli input vengono utilizzati
- cerca di scoprire il maggior numero di difetti e di escludere quelli più critici

BlackBox

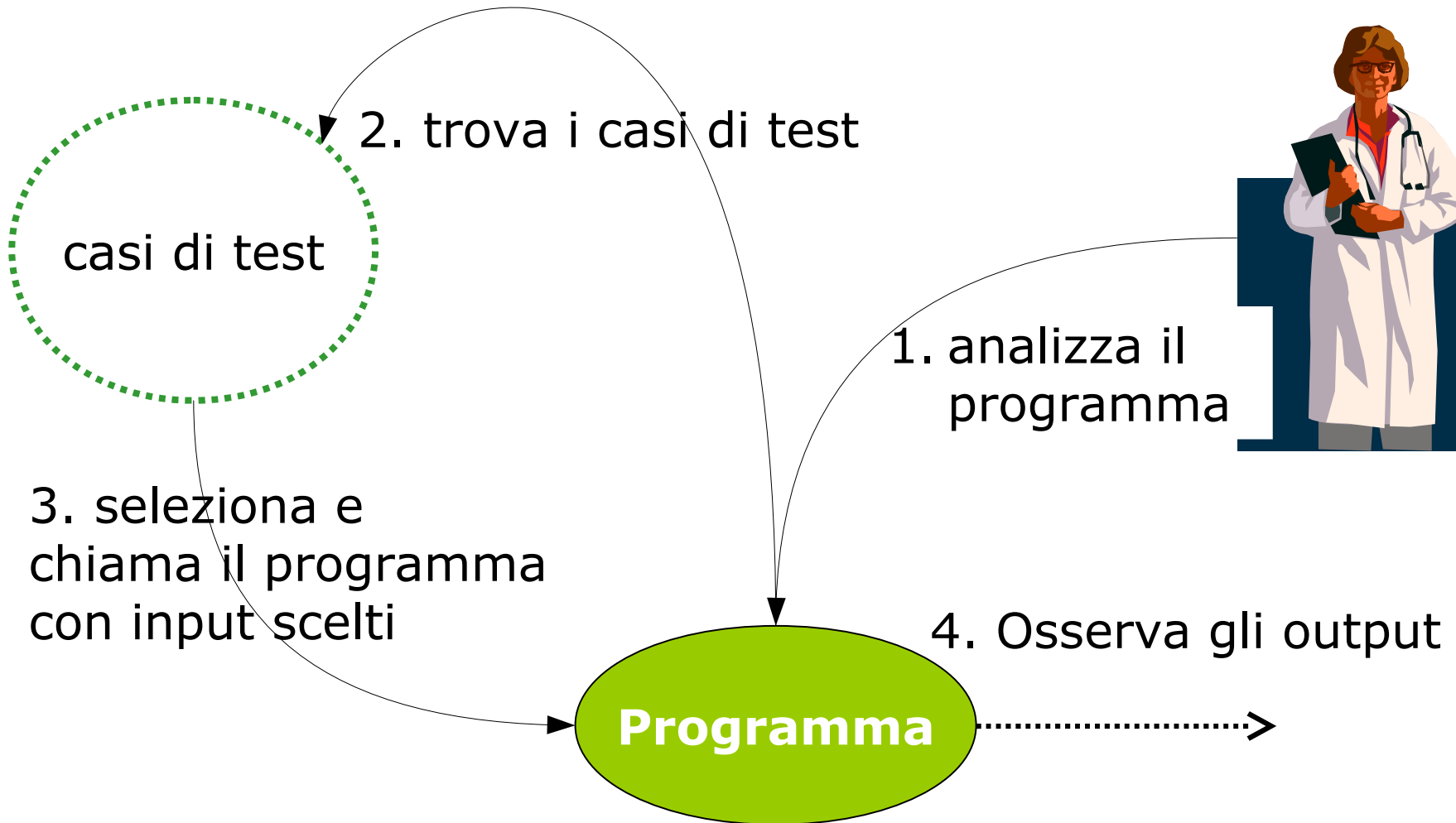


Program-Based Testing (1)

Passi principali

1. esamina la struttura del programma
 - quali i punti critici, le decisioni importanti,...
1. trova i casi di test (cioè input) che soddisfano un certo criterio di copertura
2. applica gli input (uno alla volta) e osserva il programma (output)
3. controlla che non si verifichino errori e che gli output siano quelli attesi

Program-Based Testing (2)



Valutazione del program-based testing

Vantaggi:

- il codice è una importante fonte di informazione disponibile e usabile

Limitazioni:

- non riesce a trovare errori di omissione
 - cosa succede se un programma dimentica di gestire un caso particolare? Guardando solo la struttura tale caso non verrà mai selezionato come caso di test
- non fornisce “test oracles”
 - come faccio sapere se l’output ottenuto è quello atteso?

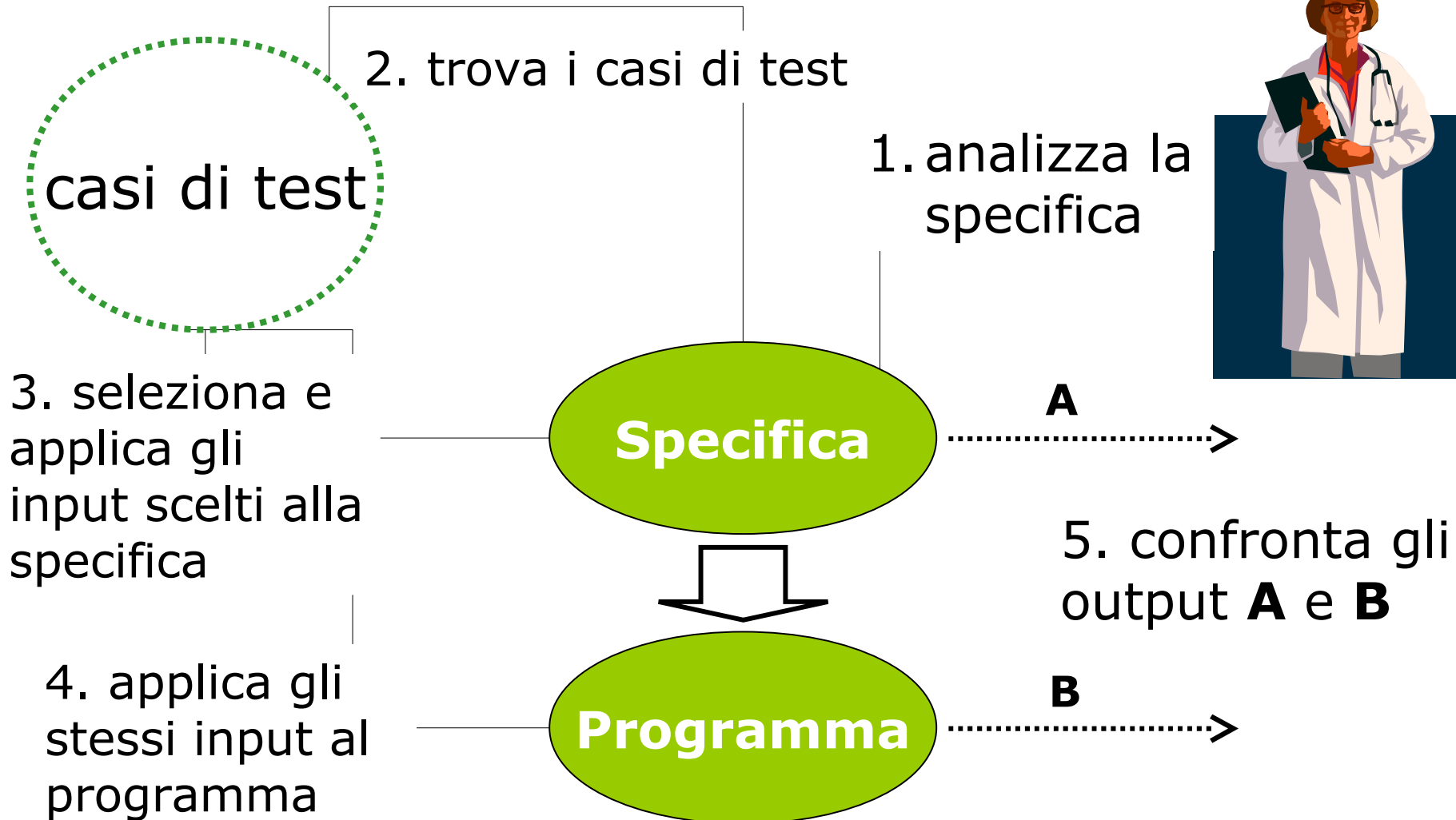
Test oracle: modo per stabilire se il test ha evidenziato un malfunzionamento oppure no

Specification-Based Testing (1)

Passi principali:

1. esamina la specifica dei requisiti del programma
2. seleziona un insieme di casi di test che soddisfano qualche criterio
3. applica questi input alla specifica e colleziona gli output **A** (**attesi**)
4. applica gli **stessi** input al programma e colleziona gli output **B** (**osservati**)
5. confronta gli output **A** con gli output **B** e controlla che siano uguali

Specification-Based Testing (2)



Valutazione del spec.-based testing

Principali vantaggi:

- la specifica funziona da **oracolo**

Principali limiti:

- le specifiche potrebbero non essere disponibili
 - potrebbe esserci solo il codice
- le specifiche devono essere “formali”
 - per generare i casi di test
 - per eseguire i casi di test selezionati per ottenere gli output
- maggiore sforzo rispetto al program-based test

In sintesi

- Abbiamo visto:
 - i diversi aspetti da testare nel sw
 - i diversi tipi di test a seconda dell'accessibilità
- Ricordate che:
 - nel white box o program based testing
 - viene analizzato solo il programma
 - si selezionano alcuni input e
 - si osserva il comportamento del programma
 - nel black box testing o specification-based
 - si tiene in considerazione la specifica di un programma
 - per generare i casi di test
 - per calcolare gli output attesi
 - da confrontare con quelli ottenuti applicando gli stessi input al programma



L1.4 Tecniche per la validazione e la verifica

Angelo Gargantini

Testing lungo il ciclo di sviluppo

Il testing si dovrebbe applicare lungo tutto il ciclo di vita del sw

Beneficio non solo della sicurezza ma anche dell'affidabilità, dell'usabilità, ...

Testing in senso allargato come **validazione e verifica o controllo del software (sw auditing)**

V&V: Validazione e Verifica

Validazione (convalida): controlla che il sw sia corretto

- accertamento che un software soddisfi i requisiti dell'utente intesi come i suoi bisogni
- si confronta con i requisiti informali espressi dall'utente
- *building the right system*

Verifica: controlla che l'implementazione corrisponda alla sua specifica

- *building the system right*

Validazione del progetto

Errori nei progetti hanno un notevole impatto

- la correzione di tali difetti richiede un notevole sforzo (ad esempio se il sw è stato già sviluppato e rilasciato)
- trovare ed eliminare bug fin dall'inizio è molto importante

Cattive notizie:

- la validazione di progetti è poco supportata, possiamo dare solo linee guida procedurali
- alcuni tool e strumenti di supporto esistono ma richiedono un grande sforzo (es. i metodi formali)

Linee guida per la validazione di progetti

Effettua una revisione del progetto

- indipendentemente da quanto formale sia il progetto, andrebbe rivisto prima di passare all'implementazione
 - usa diagrammi di flusso e altri diagrammi
 - pensa a cosa un utente malevolo potrebbe fare

Usa revisori esterni

- coinvolgendo altre persone non direttamente coinvolte nel progetto, aumenta la possibilità di trovare errori

Usa checklist e scorecard

- le checklist contengono una lista di controlli e scorecard sono simili ma assegnano anche un punteggio

Uso dei metodi formali

Valuta l'uso di metodi formali

- il progetto (o parte) potrebbe essere scritto e verificato mediante **metodi formali**
- si scrive un **modello** del sistema
- linguaggi e notazioni con una forte base logico-matematica
 - **esempio:** per una centrale nucleare, la logica di controllo potrebbe essere (anzi è) scritta mediante metodi formali per poi dimostrare effettivamente che se succede un evento pericoloso la centrale viene fermata
- tecniche molto potenti ma costose e che richiedono un grosso investimento
- **Esempi:** Z, Abstract State Machines, B, ...

Tecniche di analisi di specifiche formali

I metodi formali permettono l'uso di queste tecniche:

analisi sintattiche e di conformità

- la specifica deve soddisfare alcuni vincoli per essere considerata "ben scritta"
 - esempio: **completa** – ogni evento occorra è detto formalmente cosa succede; **consistente** – la reazione ad un evento è detta in modo non ambiguo e unico

verifica formale di proprietà

- **dimostrazione** formale che le proprietà desiderate sono implicate o soddisfatte (come logica) dal modello

simulazione e esecuzione di scenari

- **esecuzione** (interattiva e grafica) di scenari o casi d'uso per essere sicuri che la specifica catturi i requisiti dell'utente

Validazione a livello di implementazione

Le tecniche di V&V d'implementazione si dividono in

Statiche: collezionano informazioni su sw senza eseguirlo

- revisioni, walkthroughs, **ispezioni**, ...
- analisi statica automatica del codice
- verifica formale di correttezza di programmi

Dinamiche: analizzano informazioni sul sw eseguendolo

- testing: esecuzione di scenari per la ricerca di difetti
 - random testing o con criteri
- profiling: valutazione delle prestazioni

Analisi statica del codice

Analisi del flusso di controllo e del flusso dei dati

- usata già dal compilatore per ottimizzare
- valutazione di qualità del codice (non solo correttezza)

Esempi

- istruzioni non raggiungibili
- variabili usate prima di essere inizializzate
- variabili dichiarate e mai usate
- variabili scritte e mai lette
- variabile mai usata tra due assegnamenti
- possibili violazioni di array

Esistono diversi tool per l'analisi statica del codice, tra cui (open source) per C

RATS:

Rough Auditing Tool for Security

http://www.securesw.com/download_rats.htm

SPLINT:

Annotation-Assisted Lightweight Static Checking

www.splint.org

FLAWFINDER:

<http://www.dwheeler.com/flawfinder/>

Verifica formale di programmi (1)

Un programma si può formalmente provare corretto usando la logica

sw **corretto**: vuol dire che realizza esattamente le sue specifiche che devono essere date esattamente e formalmente

- metodo molto potente, garantisce che il sw non contiene difetti
- richiede normalmente grande sforzo nella prova e nella scrittura della specifica

Verifica formale di programmi (2)

- La logica più usata è la **logica di Hoare** che permette di esprimere le specifiche del sw mediante precondizioni e postcondizioni
- Normalmente la prova viene condotta **a mano** con l'ausilio di strumenti automatici
- Esistono dei metodi completamente automatici (model checkers) per eseguire la prova di correttezza

Esistono numerosi tools sperimentali

- BLAST <http://embedded.eecs.berkeley.edu/blast/>
- Key <http://www.key-project.org/>
- SLAM <http://research.microsoft.com/slam/>

Ispezione, poco tecnologica, ma efficace

Ispezione di codice secondo il modello Fagan

- una delle molte tecniche di ispezione, tra le tecniche di maggior successo
 - più formale e ben strutturata di molte altre proposte
- è estensibile a progetto, requisiti, ... seguendo principi organizzativi analoghi

Una tecnica completamente manuale per trovare e correggere errori

Ruoli nell'ispezione di software

Moderatore:

- tipicamente proviene da un altro progetto; presiede le sedute, sceglie i partecipanti, controlla il processo

Lettori, addetti al test:

- leggono il codice al gruppo, cercano difetti

Autore:

- partecipante passivo; risponde a domande quando richiesto



Checklists – esempio NASA

Circa 2.5 pagine per codice C , 4 per FORTRAN

- diviso in: funzionalità, uso dei dati, controllo, connessioni, calcolo, manutenzione, chiarezza

Esempi:

- ogni modulo contiene una singola funzione?
- il codice corrisponde al progetto dettagliato?
- i nomi di costante sono maiuscoli?
- si usa il cast di tipo dei puntatori?
- "INCLUDE"annidati di files sono evitati?
- gli usi non standard sono isolati in sottoprogrammi e ben documentati?
- i commenti sono sufficienti per capire il codice?

Ispezione - attenzione

Difetti trovati nell'ispezione non sono usati nella valutazione personale

- i programmatori non hanno motivo di nascondere i difetti

Difetti trovati durante il test (dopo l'ispezione) sono usati nella valutazione personale

- i programmatori sono incentivati a trovare difetti nell'ispezione, ma non inserendone intenzionalmente

Perchè l'ispezione funziona?

L'evidenza dice che è cost-effective, perchè?

- Processo formale, dettagliato con tracciamento dei risultati
- Check-lists: processo che si automigliora
- Aspetti sociali del processo, specialmente per gli autori
- Si applica anche a programmi incompleti

Limiti

- Scala: tecnica inerentemente a livello di unità
- Non incrementale

Runtime checkers

- analizzano l'esecuzione del codice per vedere se ci sono violazioni di memoria, deadlocks, ... prima di passare le chiamate al SO
- esempi: LibSafe, PurifyPlus, Immunix tools

Profilers

- evidenziano comportamenti anomali dell'applicazione (non necessariamente scorretti)

Penetration testing tools

- controllano (scan) l'applicazione secondo schemi di vulnerabilità conosciuti

Abbiamo visto:

- **validazione** consiste nel controllo che il sw sia corretto rispetto ai bisogni dell'utente
- **verifica**: che il sw corrisponda alla sua specifica

Ricordati che:

- per la validazione di progetto ci sono alcune linee guida
 - e tecniche come i **metodi formali**
- per la validazione e verifica dell'implementazione ci sono alcune tecniche di analisi del codice:
 - **analisi dinamica** che richiede l'esecuzione
 - **analisi statica**, tra cui l'ispezione e i check list

