



UNIVERSITÀ DEGLI STUDI DI NAPOLI  
**FEDERICO II**

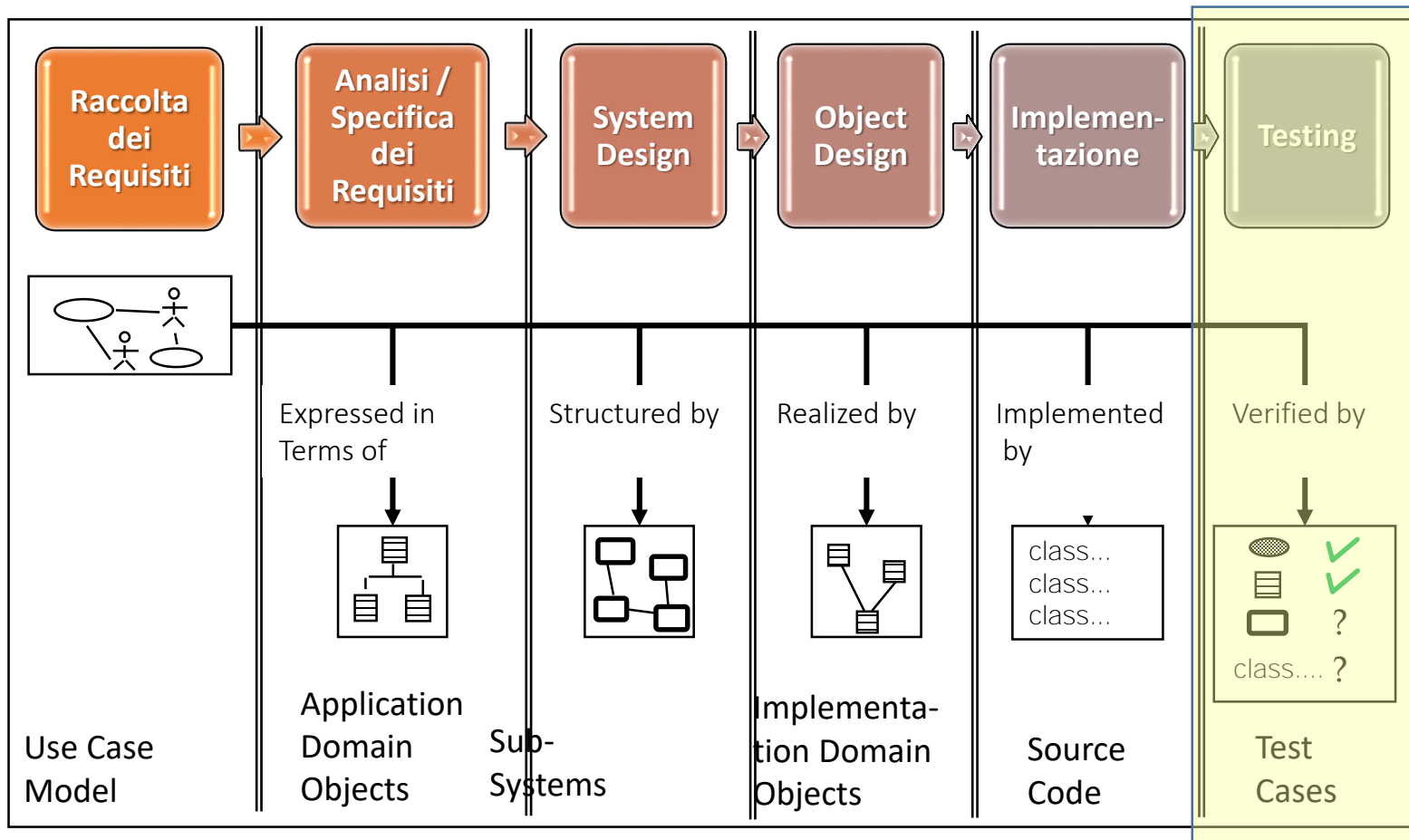
# Ingegneria del Software – Verifica e Validazione

Prof. Sergio Di Martino

# Obiettivi della lezione

- Comprendere i concetti di Verification & Validation
  - Terminologia
- Tecniche per aumentare l'affidabilità del codice
  - Inspection
  - Testing
    - I livelli di testing
    - Le strategie di testing

# Ciclo di Vita del Software



# Verification & Validation

- “The software is done. We are just trying to get it to work...”
- Finito di scrivere il codice, possiamo dire di aver finito lo sviluppo del software?
- Guardando il passato, qualche esempio ci dice che le cose non stanno proprio così...
  - Gli errori possono succedere in ogni fase del ciclo di vita!

# I bugs...

- Nella storia dell'informatica ci sono innumerevoli casi di sistemi con comportamenti che deviano dalle specifiche
- Primo caso di bug (e debug) della storia:
  - Una falena (bug) entrò nel Pannello F, Relay #70 del computer Harvard Mark II, nel 1945, creando un cortocircuito. Il computer iniziò a sbagliare addizioni e moltiplicazioni.
  - La falena (ben cotta?) fu presa (de-bug), ed attaccata sul logbook del computer con dello scotch, con la scritta "primo caso di bug trovato" ("first actual case of a bug being found")

92

9/9

0800 Antam started  
 1000 " stopped - antam ✓  
 1300 (032) MP - MC ~~1.582647000~~  
 (033) PRO 2 2.130476415  
 connect 2.130676415

Relays 6-2 in 033 failed special speed test  
 in relay " 10.00 test.

Relay  
 2145  
 Relay 3376

1100 Relays changed  
 Started Cosine Tape (Sine check)  
 1525 Started Multi-Adder Test.

1545



Relay #70 Panel F  
 (moth) in relay.

First actual case of bug being found.  
 1630 Antam started.  
 1700 closed down.

# Qualche “errore” del passato

- Errori di progettazione
- NASA Mars Climate Orbiter
  - La sonda spaziale Mars Climate Orbiter, costata 125 milioni di dollari, fu persa il 23 settembre 1999, mentre entrava nell'orbita di Marte.
  - [ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO\\_report.pdf](ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf)
  - Si scoprì successivamente che la causa era in un errore insito nelle specifiche del software adottate dalla NASA.
  - Durante il processo di sviluppo, incredibilmente, gli analisti non avevano specificato il sistema di riferimento da usare per la gestione dei motori usati nell'atterraggio
  - Come risultato, un team di sviluppo utilizzò il sistema Imperiale (pounds), mentre un altro il sistema Metrico (Kg)
  - Quando furono passati i parametri da un modulo sw all'altro, non fu operata conversione dei dati, con il risultato di dare direzioni di volo completamente errate, che portarono allo schianto della sonda.

# Qualche “errore” del passato

- Errori di coding
- ESA Ariane 5
  - Il vettore spaziale Ariane 5 esplose durante il suo volo inaugurale, il 4 Giugno 1996.
  - <http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf>
  - Per la gestione dei motori, l'Ariane 5 utilizzava il codice funzionante correttamente sul suo predecessore Ariane 4.
  - Tra questi, utilizzava una routine aritmetica di conversione da un floating point a 64bit ad un intero a 16 bit.
  - I valori generati dai motori dell'Ariane 4 erano gestibili da interi a 16bit.
  - I motori dell'Ariane 5, più potenti, generavano valori più grandi.
  - Il conseguente overflow durante la conversione portò al crash sia del sistema primario che secondario di navigazione, con la conseguente esplosione del razzo.



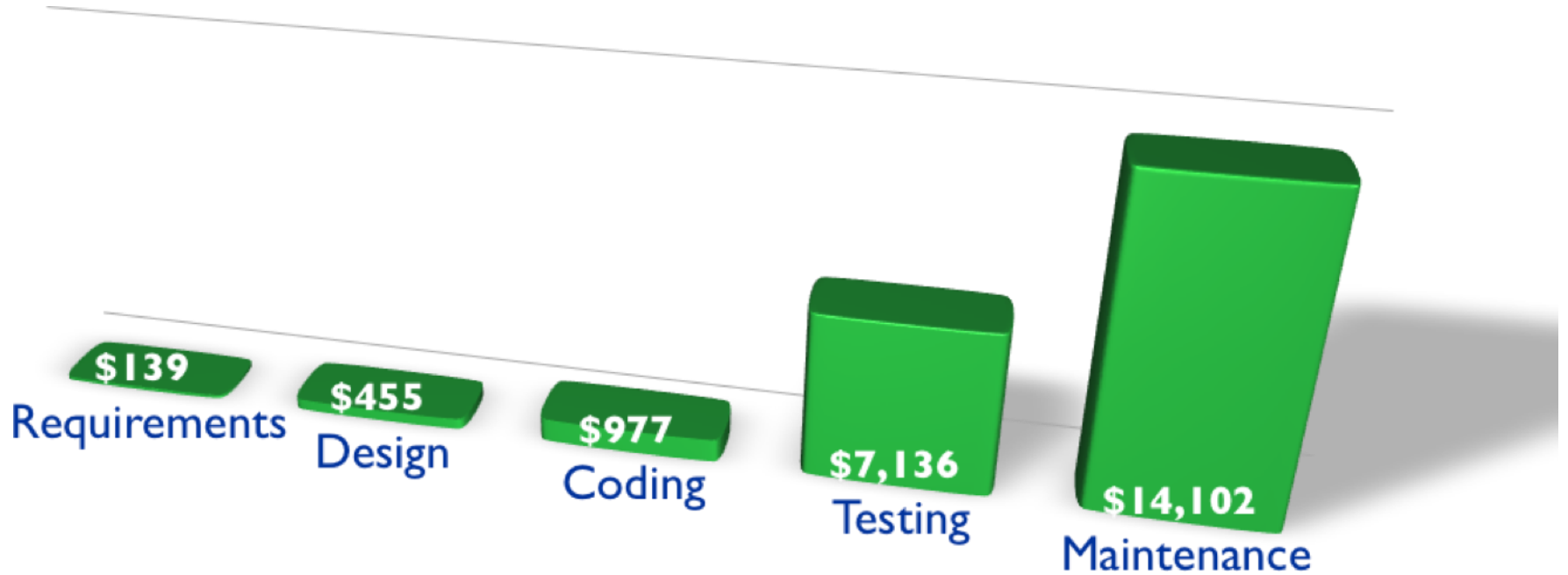
# Qualche “errore” del passato

- Errori nella gestione dell'input dell'utente
- USS Yorktown CG-48
  - Durante un'esercitazione in mare aperto, un sottoufficiale dell'incrociatore lanciamissili USS Yorktown inserì per sbaglio uno zero in un form del Data Base Manager di bordo.
  - Ciò portò ad una divisione per zero.
  - L'errore bloccò tutti i terminali della LAN della nave, arrivando allo shutdown di tutti i sistemi di bordo, comprese le turbine.
  - La nave restò completamente immobilizzata per ore, in pieno oceano, perché un form per l'inserimento di dati in un DB non verificava la correttezza dell'input inserito dall'utente.

# Qualche “errore” del passato

- Errori di valutazione sull'utilizzo del sistema
- NASA Spirit Rover
  - Il robottino per l'esplorazione di Marte “Spirit” improvvisamente cessò di comunicare con la Terra il 21 Gennaio 2004, qualche giorno dopo l'atterraggio.
  - Nei giorni successivi, i tecnici della NASA riuscirono a scoprire che il robot era in Fault mode, e si riavviava in continuazione.
  - Poiché l'errore si ripresentava ad ogni reboot, i tecnici capirono che era un problema della Flash Memory, o una rottura hardware.
  - Isolando la Flash attraverso un comando speciale, il robot si riavviò senza problemi.
  - Dopo 3 giorni, il team capì l'errore: il robot aveva loggato i principali parametri sia durante il volo fino a Marte, che durante i test al suolo. Come risultato, aveva creato un enorme numero di files, che portava al crash del modulo che si occupava di gestire il File System.
  - Dopo aver riformattato la Flash Memory, il robottino ha funzionato correttamente per altri 6 anni.

# I Costi



# Verification & Validation

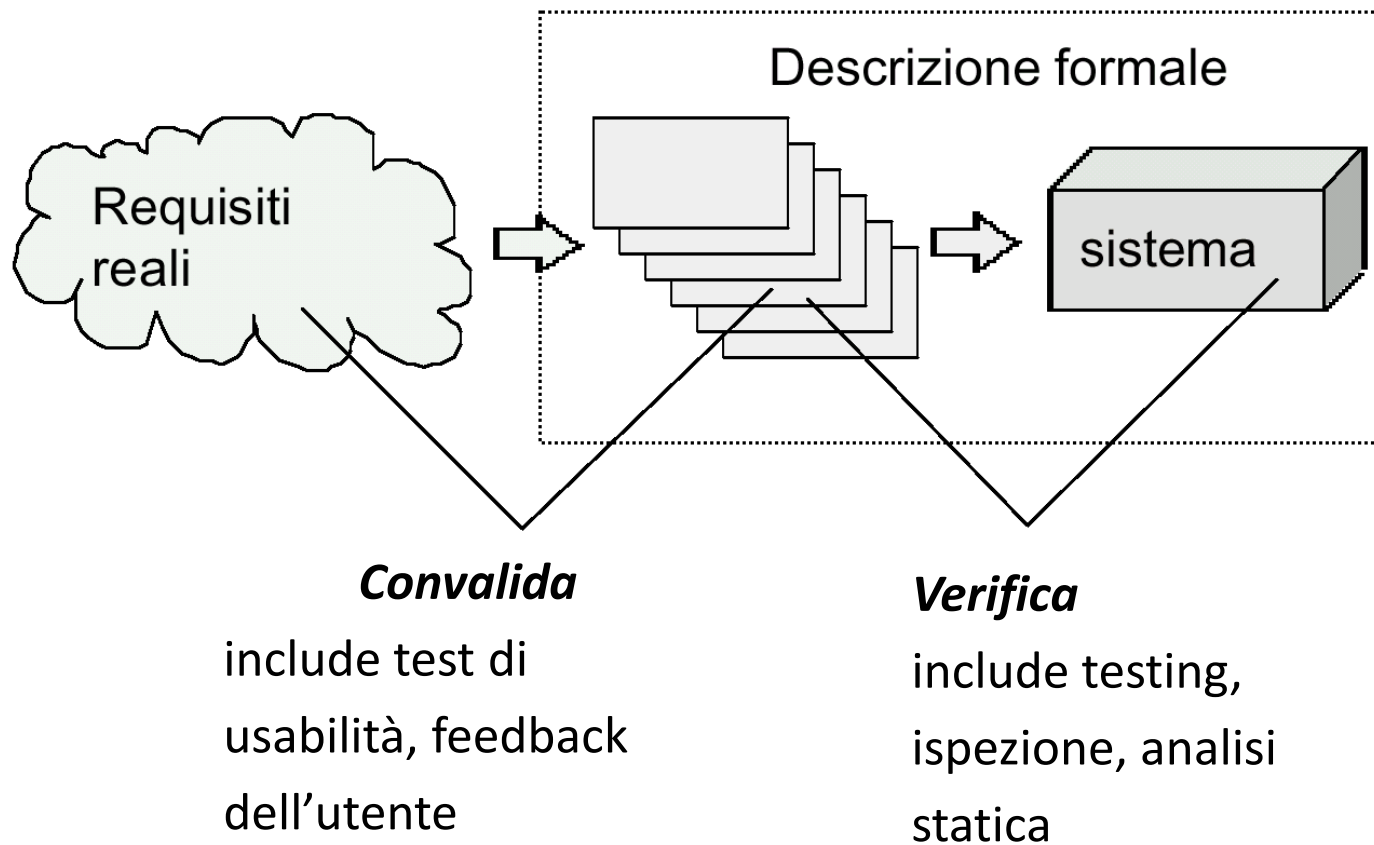
# Verification and Validation (V&V)

- [Sommerville] **Verification** and **Validation** (V&V)
  - ‘checking processes which ensure that software **conforms to its specification (at each phase in the development)** and **meets the needs of the software customer**’.
- [Boehm’79]
  - **Verification**: “Have we built the **product right**”?
  - **Validation**: “Have we built the **right product**”

# Verification and Validation (V&V)

- La fase di verifica & validazione (o convalida) serve ad accertare che il software rispetti i requisiti e che li rispetti nella maniera dovuta.
- Verifica: il software rispetta le specifiche?
  - E' stato implementato tutto quello descritto nel documento dei requisiti?
  - Ho implementato correttamente tutte le funzionalità?
  - Da sola non basta.
- Validazione: il software rispetta ciò che voleva il cliente?
  - In altri termini: i requisiti modellano ciò che il cliente realmente voleva?
  - Questa fase è molto delicata in quanto, dopo tutto il processo si può ottenere un software perfettamente funzionante, senza errori, ma del tutto inutile in quanto non rispecchia quanto era stato chiesto all'inizio.
- La verifica può essere **statica**, se effettuata senza eseguire il codice, o **dinamica**, se effettuata attraverso l'esecuzione del software.

# Verifica e Convalida



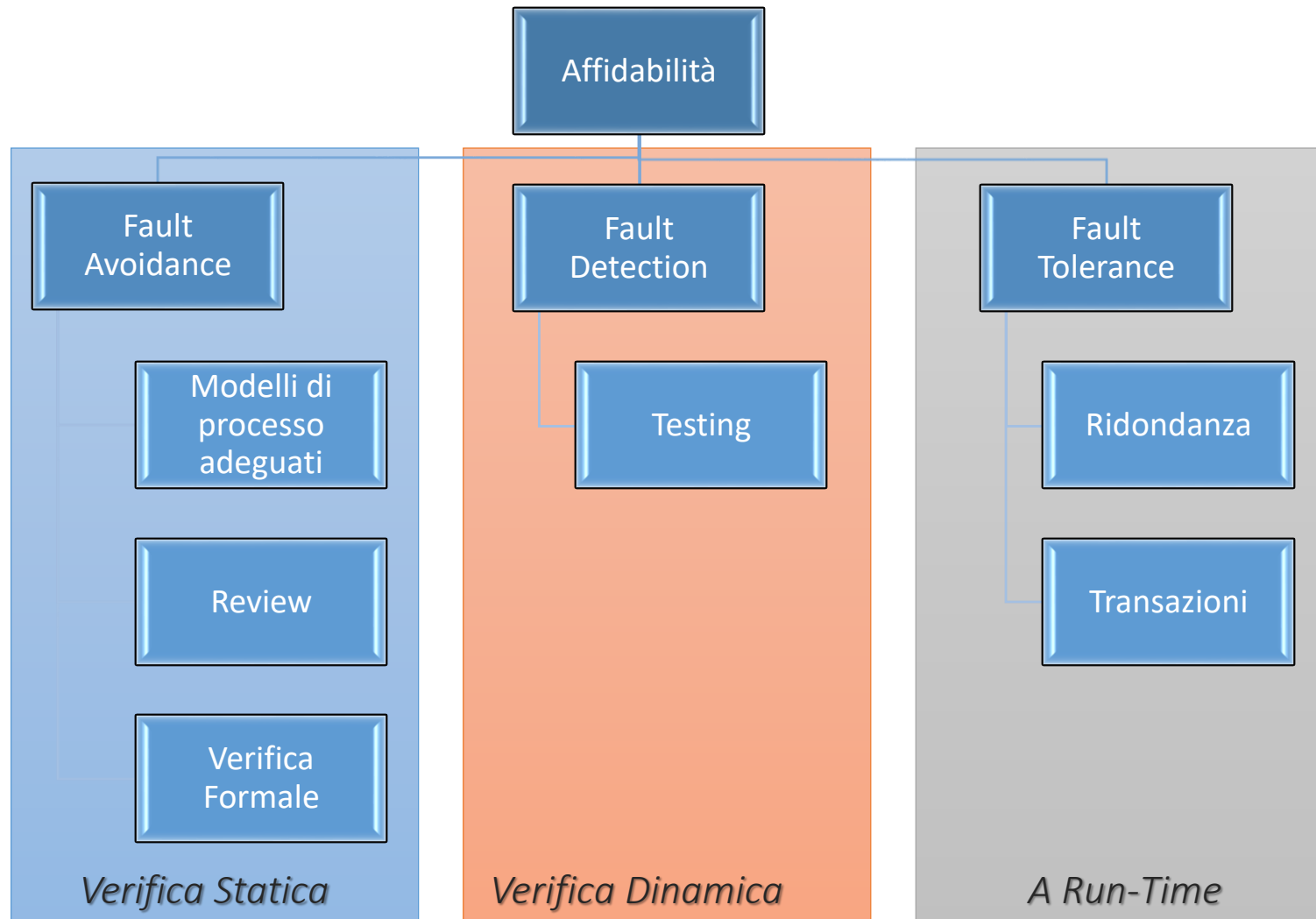
Affidabilità del sw



# Software Qualities and Process

- Qualities cannot be added after development
  - Quality results from a set of inter-dependent activities
  - Analysis and testing are crucial but far from sufficient.
- Testing is not a phase, but a lifestyle
  - Testing and analysis activities occur from early in requirements engineering through delivery and subsequent evolution.
  - Quality depends on every part of the software process
- An essential feature of software processes is that software test and analysis is thoroughly integrated and not an afterthought

# Strategie per migliorare l'affidabilità del software



# Tecniche per aumentare l’Affidabilità di un sistema software

- Le tecniche per migliorare l’affidabilità del sistema si dividono in 3 macro categorie:
  - **Fault Avoidance.** Tecniche per ridurre i difetti.
    - Tentano di prevenire l’inserimento di difetti nel sistema prima che sia realizzato
    - Includono metodologie di sviluppo, gestione delle configurazioni, e verifica
  - **Fault Detection.** Tecniche per identificare stati di errore e trovare il difetto prima di rilasciare il sistema. In molti casi sono usati anche dopo che il sistema è stato rilasciato
    - Testing: esperimenti controllati
    - Debugging: esperimenti non controllati
  - **Fault Tolerance.** Tecniche che assumono che un sistema possa contenere bug e che i fallimenti del sistema possano essere gestiti effettuando il recovery al run-time
    - Transazioni atomiche
    - Modular redundancy

# Verifica statica e dinamica

- **Software inspections:** Analisi statica delle rappresentazioni del sistema per scoprire problemi (static verification)
  - L'analisi del codice e dei documenti può essere supportata da tool
- **Software testing:** Esecuzione e osservazione del comportamento del prodotto (dynamic verification)
  - Il sistema è eseguito con dati di test

# Verifica statica

- E' basata su tecniche di analisi statica del software senza ricorso alla esecuzione del codice
  - analisi statica: è un processo di valutazione di un sistema o di un suo componente basato sulla sua forma, struttura, contenuto, documentazione
- Tecniche: ispezione, tecniche tipo compilatore, esecuzione simbolica, ecc...
- Tool come metriche di size/complexity, analisi del flusso dati
- NON E' TESTING!

# Verifica dinamica

- E' fondata sulla analisi dinamica del codice associato al software e quindi sulla esecuzione dello stesso attraverso dati di ingresso
  - analisi dinamica: il processo di valutazione di un sistema software o di un suo componente basato sulla osservazione del suo comportamento in esecuzione
- selezione di casi di test e dati associati, strumentazione del codice (es. inserzione di probe) e monitoraggio, necessità di un oracolo

# Terminologia

- **Affidabilità**: La misura di successo con cui il comportamento osservato di un sistema è conforme ad una certa specifica del relativo comportamento.
- **Fallimento** (Failure): Qualsiasi deviazione del comportamento osservato dal comportamento specificato.
- **Stato di Errore** (Errore): Il sistema è in uno stato tale che ogni ulteriore elaborazione da parte del sistema porta ad un fallimento.
- **Difetto** (Bug/fault): La causa di un errore.
- Ci sono molti tipi differenti di errori e molti modi per far fronte ad un errore.

# Fault e Failure

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

- Per il valore di ingresso  $x = 3$  si ha il valore di uscita  $y = 9$ . La causa di questa **failure** è il **fault** di linea 2, in cui anziché l'operatore  $+$  è usato l'operatore  $*$
- NB: Se il valore di ingresso è  $x = 2$ , il valore di uscita è  $y = 4$  (nessuna **failure**)



# Definizioni

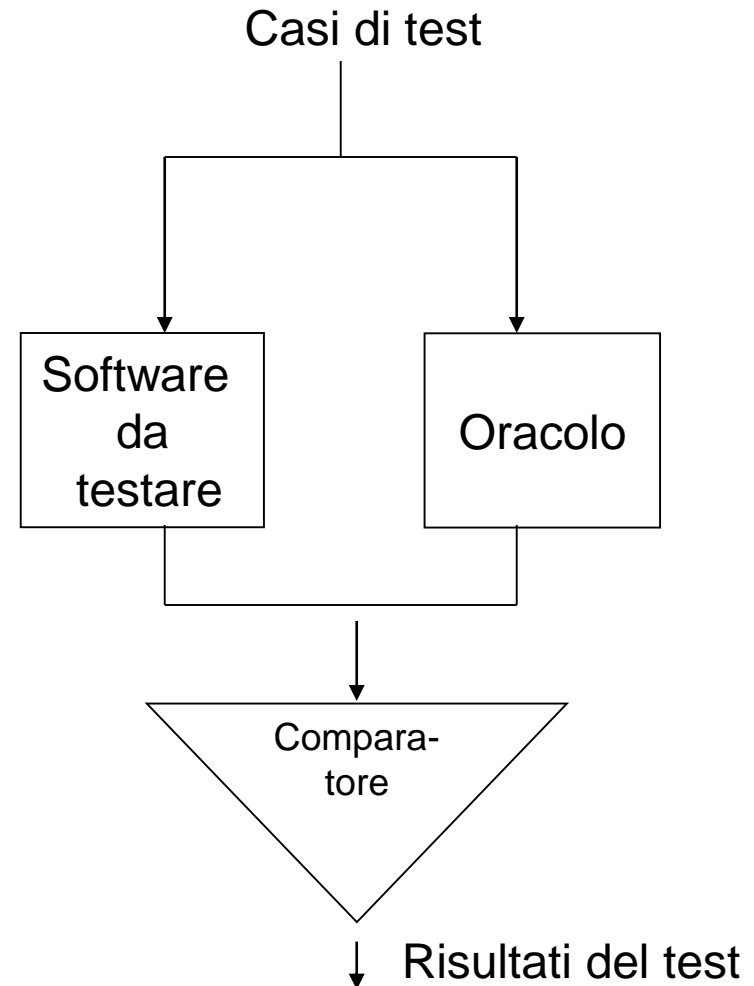
- Nota:
  - Non tutti i fault generano failure,
  - Una failure può essere generata da più fault,
  - Un fault può generare diverse failure
- Usiamo il termine **Defect** (difetto) quando non è importante distinguere fra fault e failure, per riferirsi sia alla causa (fault) che all'effetto (failure)

# Test e Casi di test

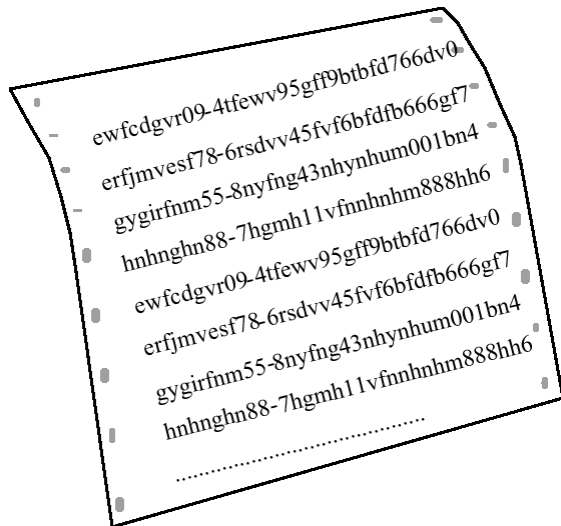
- Un programma è “stressato” da un caso di test (insieme di dati di input), o “test case”
- Un **test** è formato da un insieme di **test cases**
- L'esecuzione del test consiste nell'esecuzione del programma per tutti i casi di test
- Un test ha **successo** se rileva uno o più malfunzionamenti del programma

# L'oracolo ...

- Condizione necessaria per effettuare un test:
  - conoscere il comportamento atteso per poterlo confrontare con quello osservato
- L'Oracolo conosce il comportamento atteso per ogni caso di prova
- Oracolo umano
  - si basa sulle specifiche o sul giudizio
- Oracolo automatico
  - generato dalle specifiche (formali)
  - stesso software ma sviluppato da altri
  - versione precedente (test di regressione)



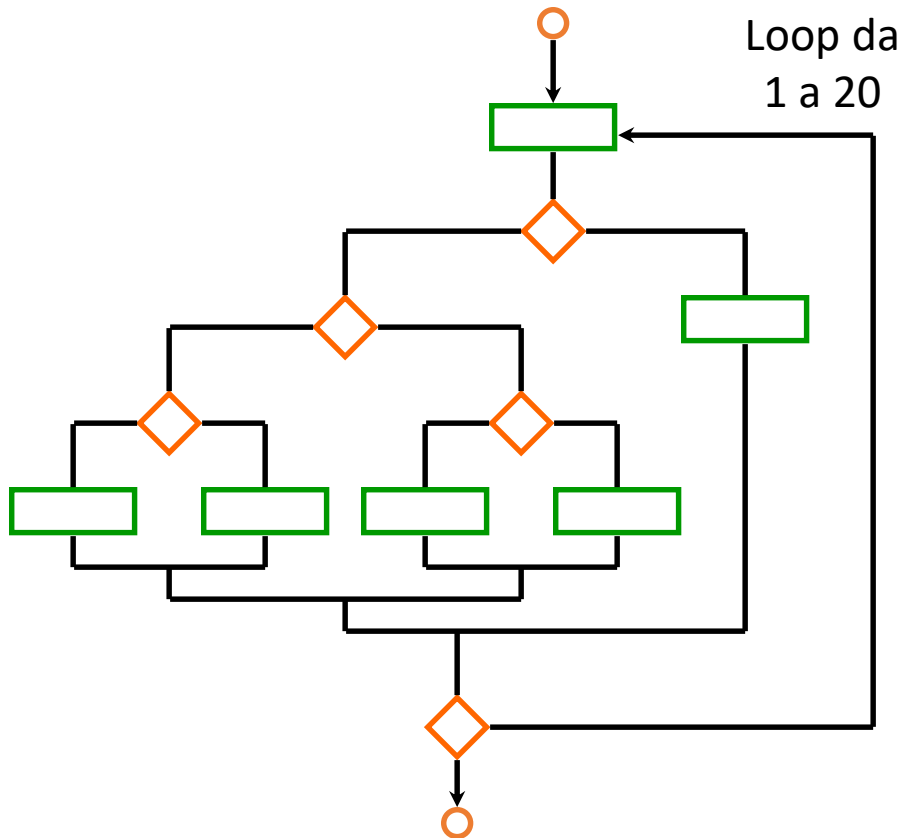
# Un buon oracolo



- Il test di applicazioni grandi e complesse può richiedere milioni di casi di test
- La dimensione dello spazio di uscita può eccedere le capacità umane
- L'occhio umano è lento e poco affidabile anche per uscite di piccole dimensioni

**ORACOLI AUTOMATICI SONO ESSENZIALI !**

# Problemi del testing: Test Esaustivo?



**$10^{14}$  cammini possibili**

**1 test / millisecondo**



**3170 anni per  
testare questo programma**

***E non e' tutto ....***

# Problemi indecidibili

- Il settore della verifica è tormentato da problemi indecidibili
  - Un problema è detto indecidibile (irrisolubile) se è possibile dimostrare che non esistono algoritmi che lo risolvono ... (macchina di Turing, halting problem, equivalenza di funzioni, etc...)
- es. stabilire se l'esecuzione di un programma termina a fronte di un input arbitrario è un problema indecidibile

# V&V goals and confidence

- Verification and validation dovrebbero stabilire un certo livello di fiducia che il software è adatto ai suoi scopi (fa quello che deve fare).
  - Questo non significa completa assenza di difetti
  - Significa invece che deve essere abbastanza buono per l'uso per il quale è stato pensato e proprio il tipo di uso determinerà il livello di fiducia richiesto
- Il livello di fiducia dipende dagli scopi e funzioni del sistema, dalle attese degli utenti e dal time-to-market

# Verifica Statica e Dinamica



# Verifica statica

- E' basata su tecniche di analisi statica del software senza ricorso alla esecuzione del codice
  - analisi statica: è un processo di valutazione di un sistema o di un suo componente basato sulla sua forma, struttura, contenuto, documentazione
- Tecniche: review, ispezione, verifica formale, esecuzione simbolica, etc...

# Verifica Statica: Le Review

- Reviews: ispezione manuale del codice sorgente
- Due tipi di review:
  - **Walkthrough.** Lo sviluppatore presenta informalmente le API, il codice, la documentazione associata delle componenti al team di review
  - **Inspection.** Simile al walkthrough, ma la presentazione delle unità è formale.
    - Lo sviluppatore non può presentare gli artefatti. Questo è fatto dal team di review che è responsabile del controllo delle interfacce e del codice con i requisiti
    - Controlla l'efficienza degli algoritmi con le richieste non funzionali
    - Lo sviluppatore interviene solo se si richiedono chiarimenti

# Analisi informali

- Analizzare la specifica dei requisiti, la specifica dei requisiti o il codice attraverso una simulazione manuale
- Code walk-through (operare come il computer)
- Ispezioni del codice – scoprire gli errori più comuni (variabili non inizializzate, salti all'interno dei cicli...)

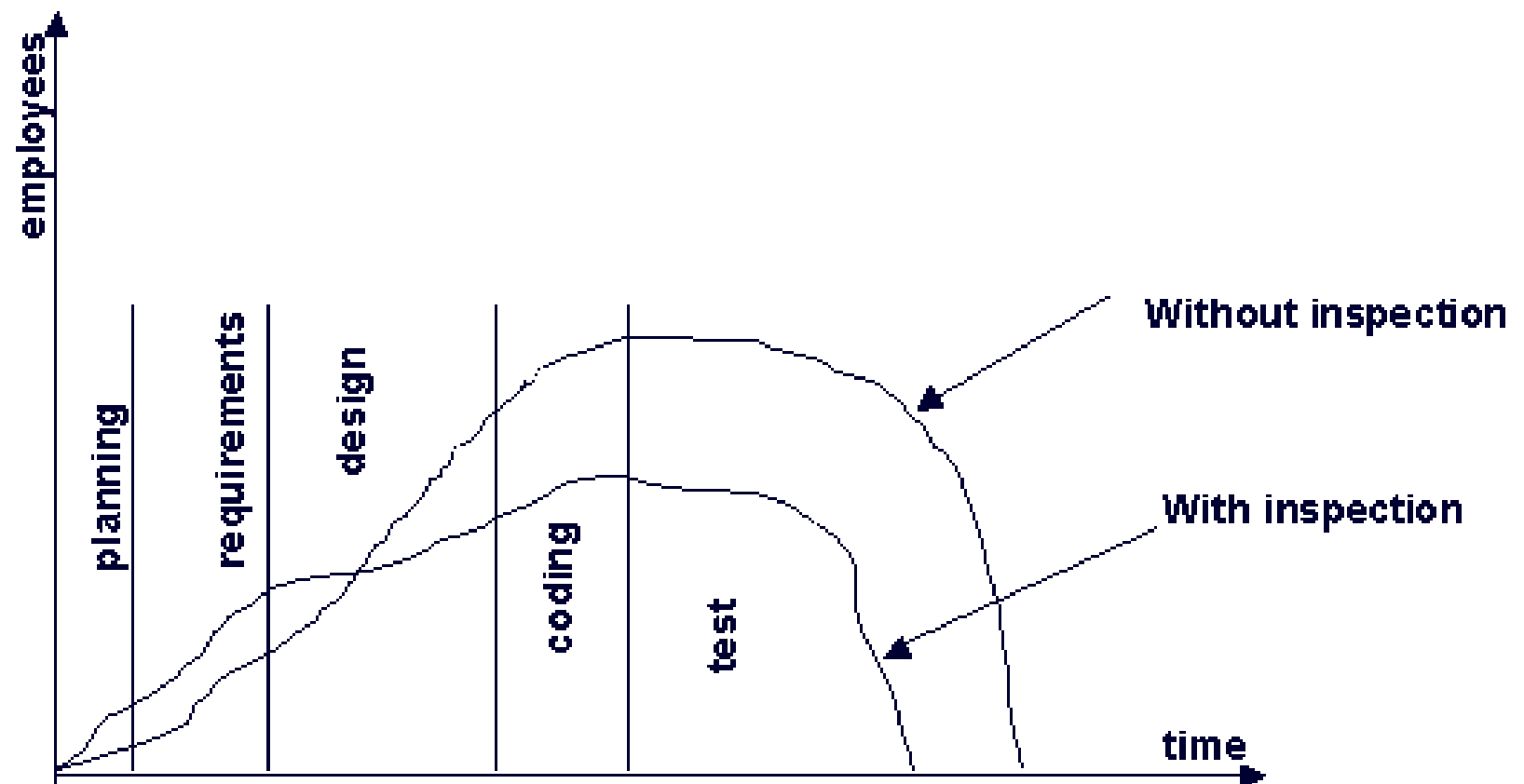
# Ispezione (Fagan, 1976)

- Le Ispezioni trovano fault in una componente rivedendo il codice sorgente in meeting formali.
- E' condotta da un team di sviluppatori, incluso l'autore della componente, un moderatore e uno o più revisori che trovano i bug nella componente
- Tecniche di ispezione di codice possono rilevare ed eliminare anomalie e rendere più precisi i risultati
  - una tecnica completamente manuale per trovare e correggere errori
    - poco tecnologica, ma efficace
    - ma sono possibili alcuni supporti automatici ...
- E' estendibile ad altri artefatti seguendo principi organizzativi analoghi

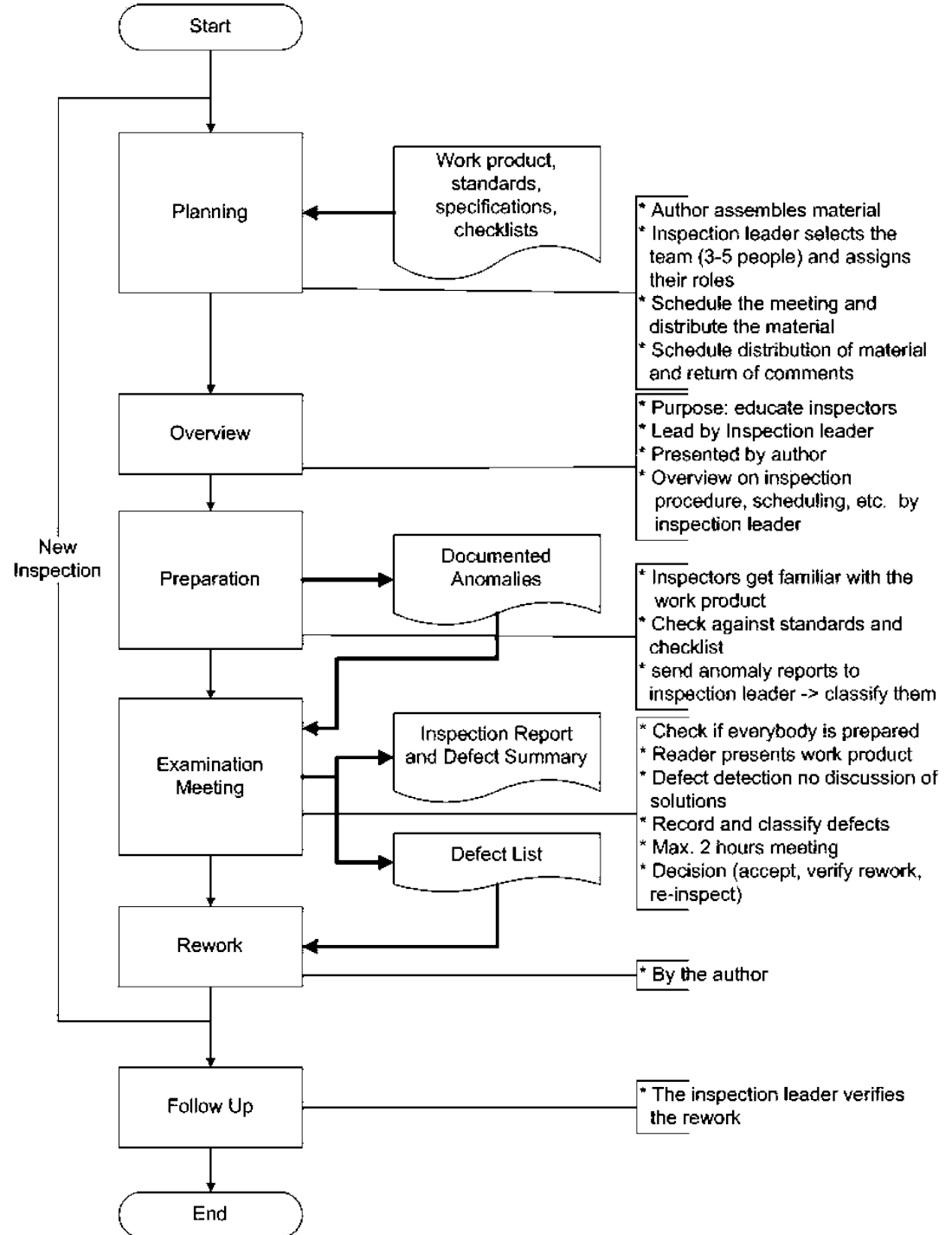
# Ispezione

- Vantaggi principali rispetto al testing:
  - Durante un test, un errore può nascondere altri errori. Ciò viene superato leggendo il codice sorgente
  - Non ho necessità di sviluppare driver-stub. Posso applicarlo a codice definitivo in unità incomplete
  - Permette di valutare anche altri attributi del software (qualità, leggibilità, aderenza a standard, etc...)

# Costi



## II Processo



# 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



# Il processo di ispezione del software

- pianificazione
  - scelta di partecipanti e checklist, pianificazione di meeting
- fasi preliminari
  - fornite le informazioni necessarie e assegnati i ruoli
- preparazione
  - lettura del documento, individuazione dei difetti
- Ispezione
  - meeting: raccolta e discussione congiunta dei problemi trovati dai singoli revisori, ricerca di ulteriori difetti
- lavoro a valle
  - l'autore modifica il documento per rimuovere i difetti
- seguito (possibile re-ispezione)
  - controllo delle modifiche, modifica delle checklist

# Nelle riunioni ...

- obiettivo: trovare il maggior numero possibile di difetti
  - massimo 2 riunioni al giorno di 2 ore ciascuna
  - circa 150 linee di codice sorgente all'ora
- approccio: parafrasare il codice linea per linea
  - ricostruire l'obiettivo dal codice sorgente
- necessario restare in tema
  - seguire le checklist
  - trovare e registrare difetti, ma non correggerli
  - il moderatore è responsabile di evitare anarchia

# Example Defect List Form

<b>Project:</b>	_____	<b>Meeting Date:</b>	_____
<b>Phase:</b>	_____	<b>Release:</b>	_____
<b>Component:</b>	_____	<b>Document:</b>	_____
<b>Preparation Time:</b>	_____	<b>Inspector:</b>	_____

[illegible]

# Inspection Defect Log

<b>Product</b>	Simple Sort	<b>Date</b>	October 23, 1999	
<b>Author</b>	Fraser Macdonald			
<b>Defect#</b>	<b>Description</b>	<b>Type</b>	<b>Location</b>	<b>Severity</b>
1	Function max() is defined, but never used. No failure apparently, but checklist violation.	function calls	line 12, function max()	trivial
2	Parameters are passed by value, not by reference. "swap" doesn't correctly swap the numbers, so the sort is not carried out correctly.	function calls	line 5, function swap()	failure

# Checklist – 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?

# Checklist per Java

- Sintassi
  - controlli ortografici, grammaticali e linguistici
  - commenti formattati per javadoc ...
- Struttura
  - ci sono dati identificativi (titolo, data, versione, autore)
  - librerie non standard usate, piattaforma Java richiesta
- Stile
  - Non c'è un uso indiscriminato di abbreviazioni
  - Non ci sono troppe ripetizioni
  - Linguaggio comprensibile e frasi leggibili
  - Il commento è presente in ogni classe
  - Il commento è ben visibile nel file
  - Le variabili sono commentate quando sono dichiarate
  - I commenti non sono dispersivi (mal disposti, interrotti e ripresi)
  - I commenti non sono esageratamente densi nel codice (compromettendo la leggibilità del codice)

# Perché l'ispezione funziona?

- Il metodo di ispezione sono molto efficaci: Circa 85% dei fault può essere individuato
- 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
  - Considera l'intero spazio di ingresso
  - Si applica anche a programmi incompleti
- Limiti
  - Scala: tecnica inerentemente a livello di unità
  - Non incrementale

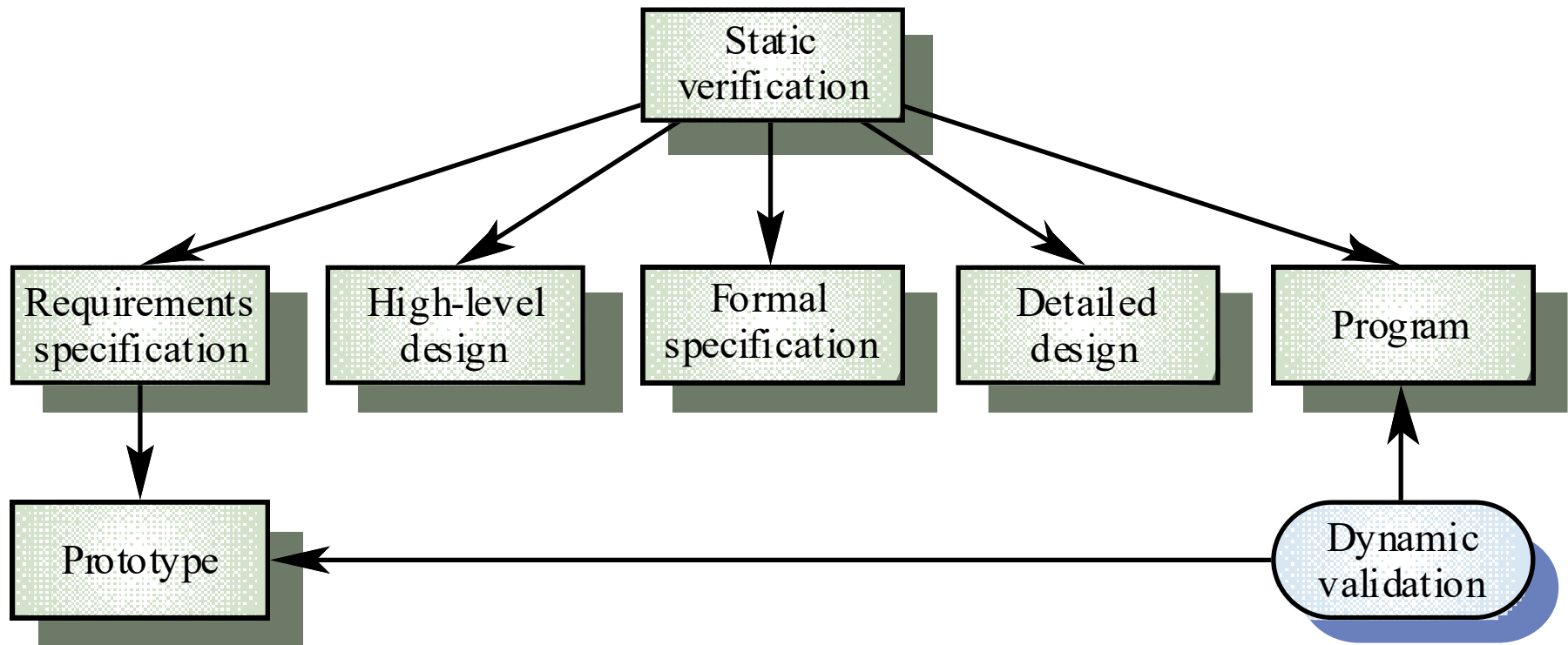
# Verifica Dinamica



# Verifica dinamica

- E' fondata sull'esecuzione del codice (testing)
  - analisi dinamica: il processo di valutazione di un sistema software o di un suo componente basato sulla osservazione del suo comportamento in esecuzione
- **Testing:** Approccio strutturato alla selezione di casi di test e dati associati
- **Debugging:** non c'è uniformità in letteratura
  - In generale, riguarda l'individuazione e l'eliminazione dei difetti
  - Debugging implica formulare ipotesi osservando il comportamento del programma e quindi verificare queste ipotesi per localizzare gli errori

# Static and dynamic V&V



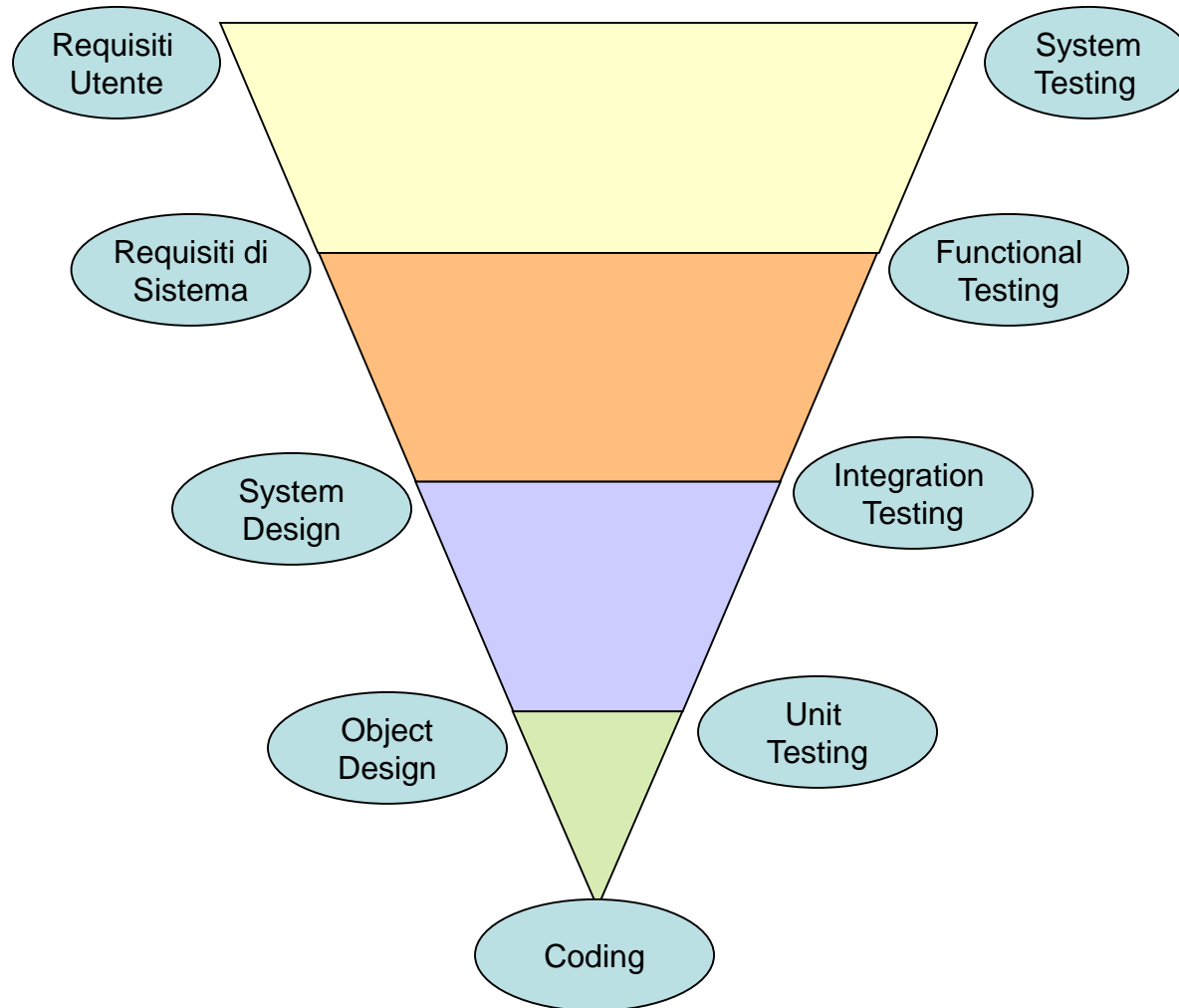
# Overview

- Il Testing consiste nel trovare le differenze tra il comportamento atteso, specificato attraverso il modello del sistema e il comportamento osservato dal sistema implementato.
- Obiettivo: progettare test per provare il sistema e rivelare problemi
  - Massimizzare il numero di errori scoperti che consentirà agli sviluppatori di correggerli
- Questa attività va in contrasto con le altre attività svolte prima: analisi, design, implementazione sono attività “costruttive”. Il testing invece tenta di “rompere” il sistema
- Il testing dovrebbe essere realizzato da persone che non sono state coinvolte nelle attività di sviluppo del sistema

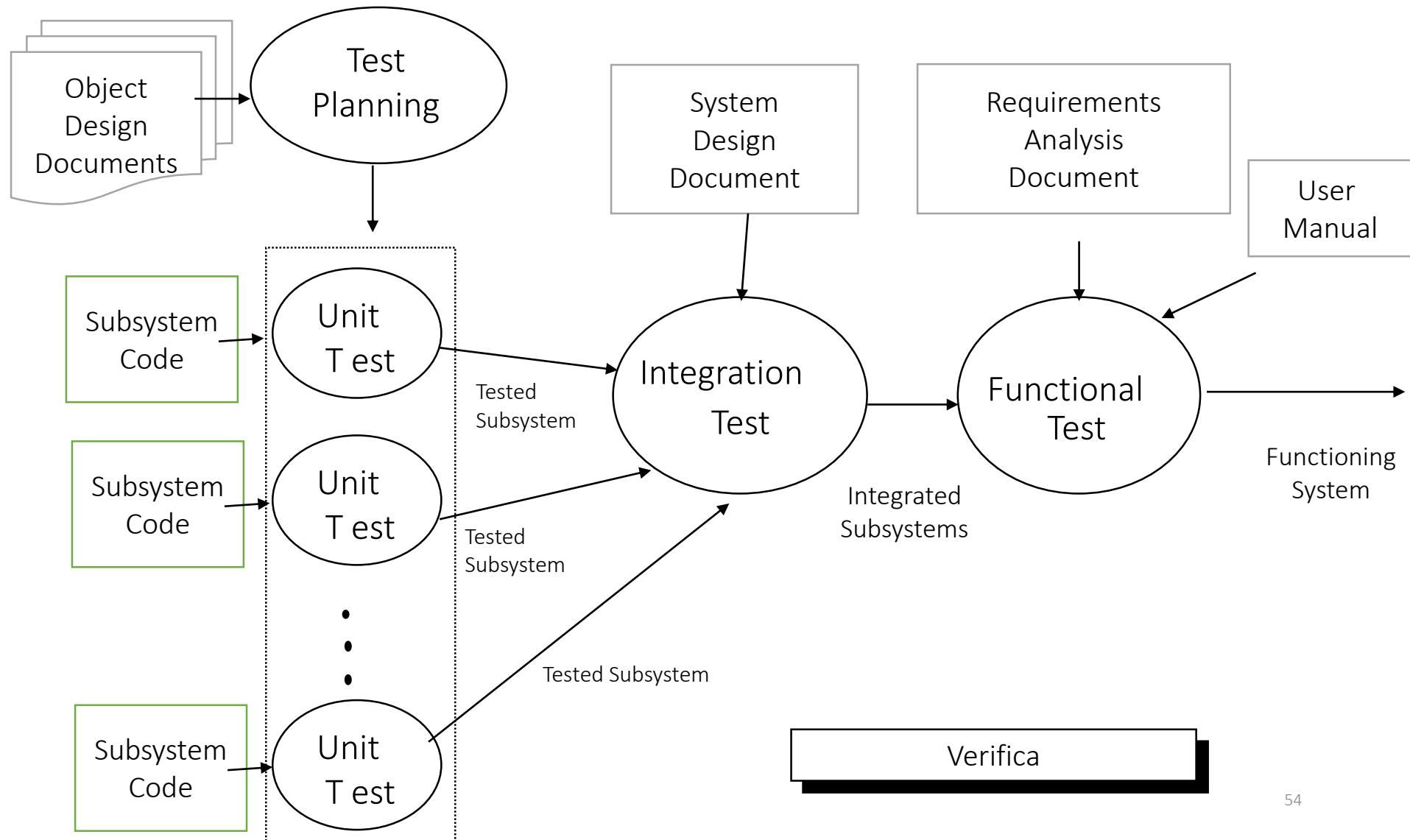
# Cosa testare?

- Il testing avviene a vari **livelli**:
  - Unit testing
    - Trovare differenze tra object design model e corrispondente componente
  - Integration testing
    - Trovare differenze tra system design model e un sottoinsieme integrato di sottosistemi
  - Functional testing
    - Trovare differenze tra use case model e il sistema
  - System testing
    - Trovare differenze tra requisiti non funzionali e il sistema

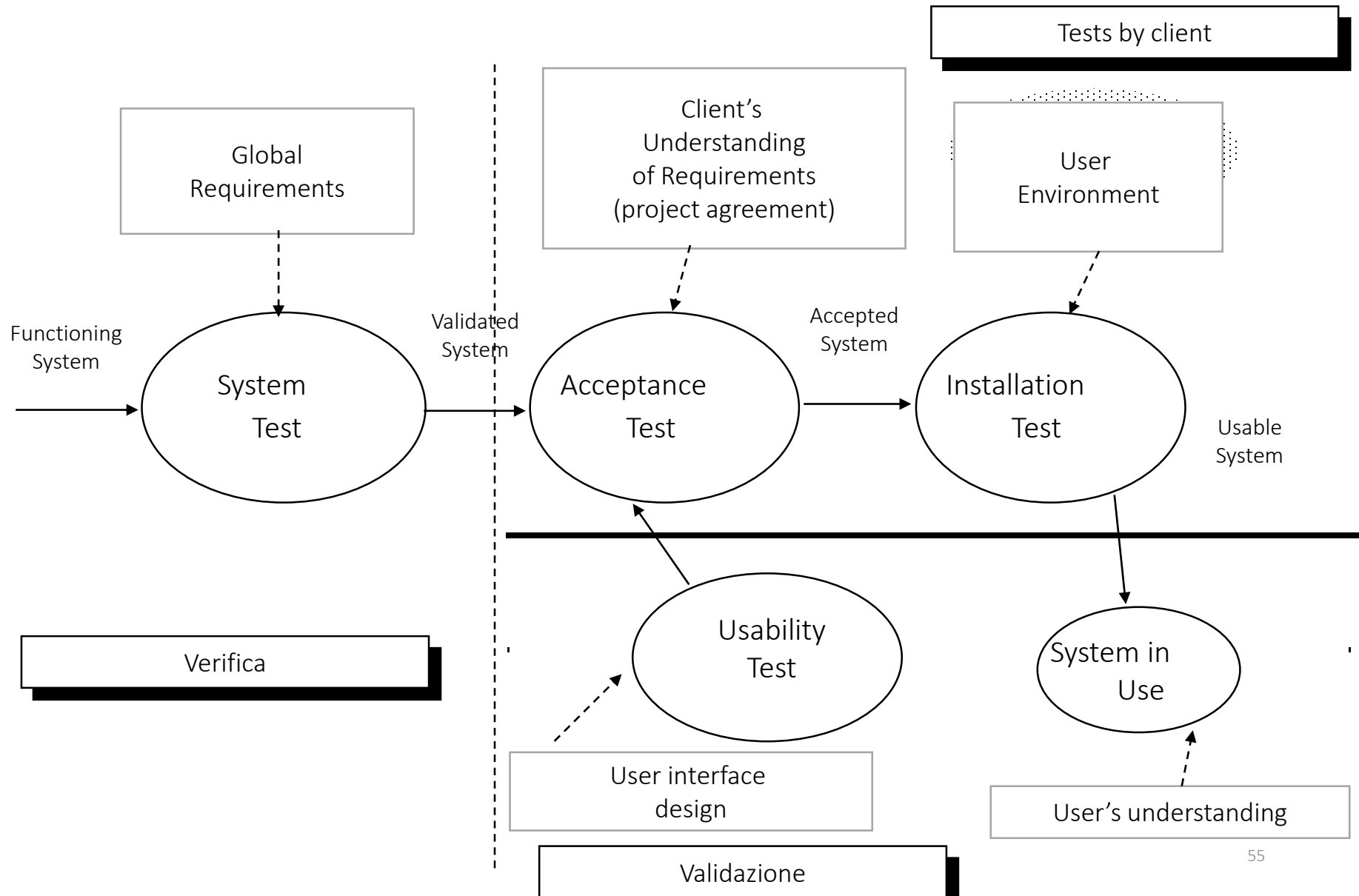
# Il modello di sviluppo a V



# Livelli di Testing



# Livelli di Testing (2)



# Unit Testing



# Testing di Unità

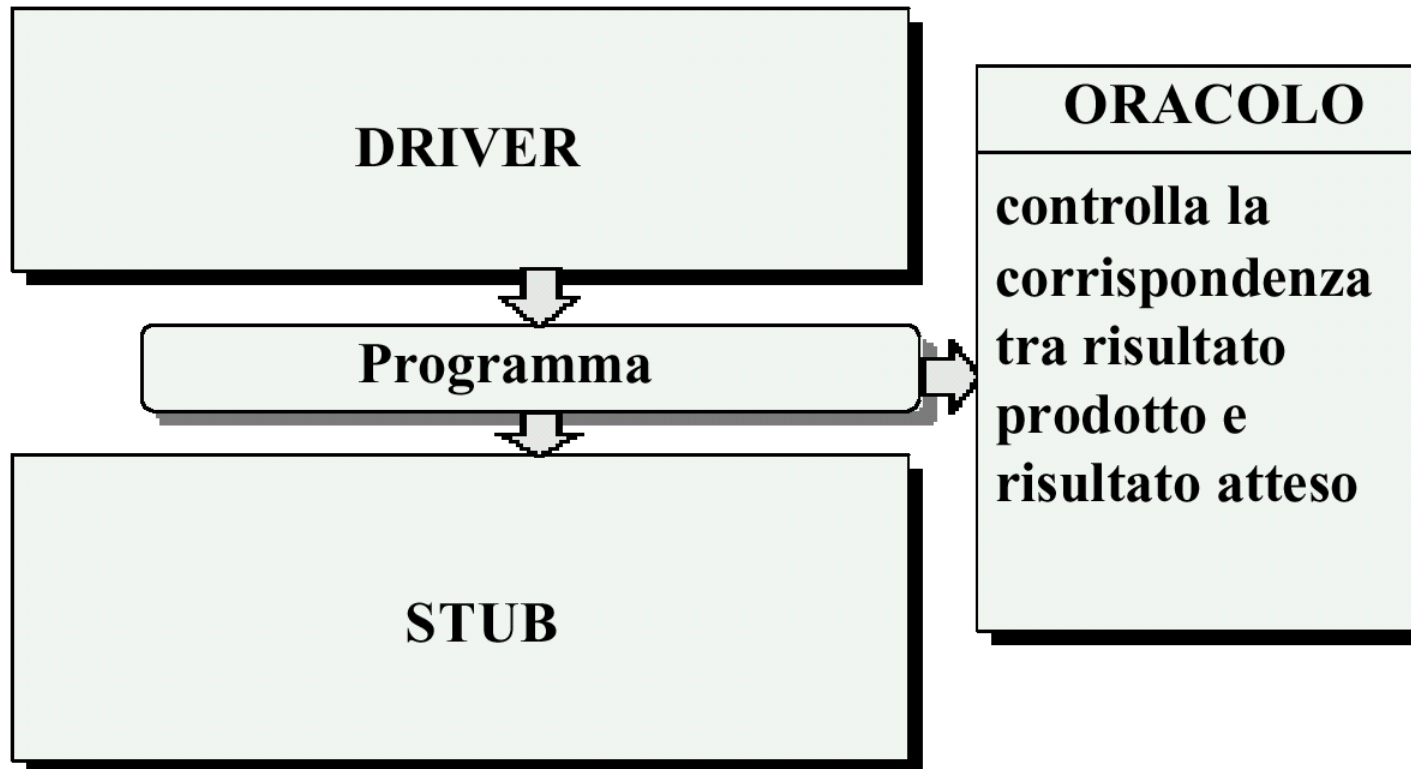
- Il testing è applicato isolatamente ad una unità (modulo) di un sistema software
- Obiettivo fondamentale è quello di rilevare errori (logica e dati) nel modulo
- Motivazioni:
  - Si riduce la complessità concentrandosi su una sola unit
  - È più facile correggere i bug, poiché poche componenti sono coinvolte

- unità: elemento definito nel progetto di un sistema software e testabile separatamente
- unità e modulo sono spesso usati come sinonimi

# Test stub e test driver

- Eseguire test case su una componente (o una combinazione di componenti) richiede che questa sia isolata dal resto del sistema
- **Test driver** e **test stub** sono usati per sostituire le parti mancanti del sistema
  - Un test stub è una implementazione parziale di componenti da cui la componente testata dipende (componenti che sono chiamate dalla componente da testare).
  - Un test driver è un blocco di codice che inizializza e chiama la componente da testare.
- **Test driver + test stub = Scaffolding**

# Creare scaffolding



*La necessità di moduli stub e di moduli driver dipende dalla posizione del modulo nell'architettura del sistema ...*

# Driver e Stub

- Driver (Modulo Guida)
  - Deve simulare l'ambiente chiamante
  - Deve occuparsi dell'inizializzazione dell'ambiente non locale del modulo in esame
- Stub (Modulo Fittizio)
  - Ha la stessa interfaccia del modulo simulato (tipicamente ne è una sottoclasse), ma è privo di implementazioni significative
  - Detti anche Mock Objects

# Il problema dello scaffolding

- Creare l'ambiente per l'esecuzione dei test
  - Lo scaffolding è estremamente importante per il test di unità e integrazione
  - Può richiedere un grosso sforzo programmatico
  - Uno scaffolding buono è un passo importante per test di regressione efficiente
  - La generazione di scaffolding può essere parzialmente automatizzato a partire dalle specifiche ...
  - Esistono pacchetti software per supportare la generazione di scaffolding
    - JUnit, NUnit, PUnit, etc...

# Implementazione di Test stub

- Un test stub deve fornire la stessa API del metodo della componente simulata e ritornare un valore il cui tipo è conforme con il tipo del valore di ritorno specificato nella signature.
  - Se l'interfaccia di una componente cambia, anche il corrispondente test driver e test stub devono cambiare
- L'implementazione di un test stub non è una cosa semplice.
  - Non è sufficiente scrivere un test stub che semplicemente stampa un messaggio attestante che il test stub è in esecuzione
    - La componente chiamata deve fare un qualche lavoro.
    - Il test stub non può restituire sempre lo stesso valore

# Progettare per il Testing

```
class Traveler
{
    Car c=new Car();
    void startJourney()
    {
        c.move();
    }
}
```

```
class Car
{
    void move()
    {
        // logic...
    }
}
```

- La classe Traveler è fortemente accoppiata con Car.
- Non è possibile passare un'istanza di Car dall'esterno.
- Diventa impossibile testare Traveler in isolamento senza fare modifiche al codice.
- Con l'introduzione dell'interfaccia Vehicle, si risolve questo problema.
- Posso definire una classe Car\_Stub, implementazione di Vehicle, e grazie alla Dependency Injection effettuata dal Test Driver, testare Traveler in isolamento.

# Esempio (cont.)

```
class Traveler
{
    Vehicle v;
    public void setV(Vehicle v)
    {
        this.v = v;
    }

    void startJourney()
    {
        v.move();
    }
}

Interface Vehicle
{
    void move();
}
```

```
class Car implements Vehicle
{
    public void move()
    {
        // logic
    }
}
```

```
class Car_Stub implements Vehicle
{
    public void move()
    {
        // Mock logic
    }
}
```