Testing

H1 Introduzione

H2

> 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

Come detto, spesso è la fase più costosa del ciclo di sviluppo.

È possibile avere un SW funzionante al 100%?
In ingegneria classica, gli oggetti da testare hanno un comportamento "continuo" (un ponte se funziona in un punto, in genere funziona tutte le sue parti). Il SW ha un comportamento molto più "discontinuo", perciò *i "punti" da testare sono critici*.

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

H3 Proprietà desiderate

Il **testing** dovrebbe:

- essere **automatizzato**
- riguardare **ogni fase di sviluppo**:
  - bisogna testare i risultati prodotti anche da fasi che non sono solo l'implementazione
- essere **esteso** a tutti i componenti di un sistema
- essere **pianificato** (*test plan*)
- seguire degli **standard** e metodologie dove possibile

Terminologia

Malfunzionamento (*failure*)

H2

> Un **malfunzionamento** è il funzionamento non corretto di un programma

H4

Un **malfunzionamento** si osserva in fase di esecuzione. (***dinamico***)

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

Con questo codice, chiamando `raddoppia(2)`, non abbiamo un **malfunzionamento**, in quanto il comportamento sembra corretto, poiché restituisce `4`. Se invece chiamiamo `raddoppia(3)`, osserviamo un **malfunzionamento**, in quanto restituisce `9` invece di `6`.

Difetto (*bug*)

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

H4

Un **bug** riguarda prettamente il codice, non la sua esecuzione. È quindi un elemento ***statico***.

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

Tornando all'esempio di codice precedente, osserviamo, senza aver bisogno di eseguire la funzione con qualche input, che vi è un **difetto**: dovrebbe essere `return 2*x`, non `return x*x`. Questo è un difetto in quanto non è quello che vorremmo dalla funzione e può portare infatti a dei **malfunzionamenti** (che come abbiamo visto prima, per certi input potrebbero non esserci, anche se il bug è presente!)

Lo scopo del **testing** è trovare **bug** attraverso **malfunzionamenti**.

Errore

H4

> Fattore (umano) che causa una deviazione tra il SW prodotto e il programma desiderato.

Si tratta quindi di tutti quegli elementi che possono generare **bug** e quindi **malfunzionamenti**, come: errore di analisi di requisiti, errore di battitura, errore di design...

Testing

H4

> Eseguire il programma con dei **casi di test** e analizzare i risultati per trovare **bug**

Gli **scopi** del testing quindi sono:

- mettere in evidenza i **bug** mediante **malfunzionamenti**, per scoprire eventuali **errori**
- poter valutare l'**affidabilità** (***reliability***) di un SW e fornire *confidenza* (**test di accettazione**)

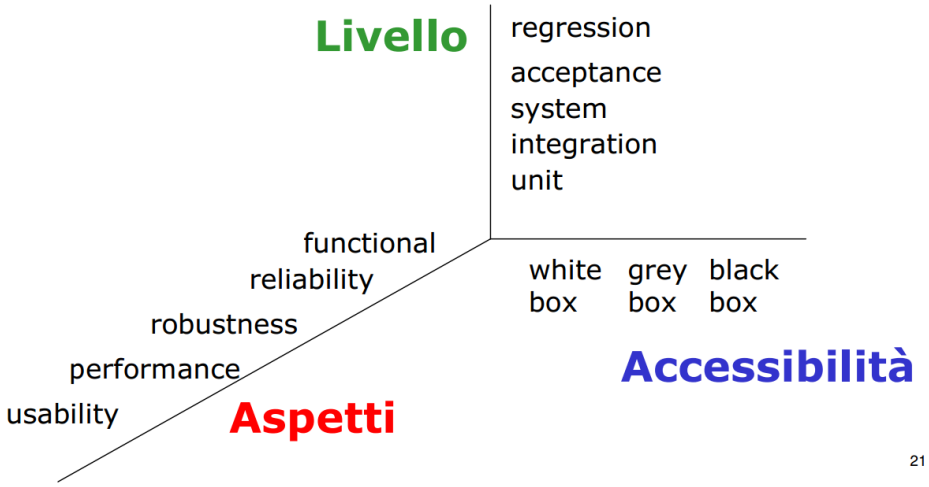
Debugging

H4

> Correggere i **bug** e eventualmente scoprire gli errori che li hanno generati

Tipi di testing

H2



Come si nota dalla figura, vi sono *diversi tipi* di **testing**, in base a 3 diverse coordinate:

- **livello**: che fase del ciclo di vita del SW si vuole testare
- **aspetti**: quali proprietà del SW si vogliono testare
- **accessibilità**: che grado di *accesso* al sistema si vuole avere

H3 Livelli

I diversi **livelli** (in ordine dal più alto al più basso) sono:
Test di:

- **accettazione**: il comportamento del SW è confrontato con i requisiti dell'utente finale
- **conformità**: il comportamento di tutto il SW è confrontato con le specifiche
- **sistema**: controlla il comportamento dell'intero sistema (HW + SW) come monolitico
- **integrazione**: controllo sul modo di cooperazione delle unità
- **unità**: controllo del comportamento delle singole unità

Inoltre, successivamente al rilascio abbiamo:

- **regressione**: test del comportamento di release successive (*manutenzione*)

Ora guardiamo nelle specifico i due livelli più bassi: **integrazione** e **unità**.

Test di Unità

H4

> Testa il codice a livello di singole unità, in OO *un'**unità** corrisponde a una classe*

Per ogni **metodo** della classe da testare, si introducono:

- **test driver**: metodo che chiama il metodo con opportuni parametri
- **test stub** (*opzionale*): metodo che sostituisce eventuali metodi usati dal metodo che si vuole testare per testarlo in modo isolato e controllato

Test d'Integrazione

H4

> Testa l'**integrazione** tra le diverse unità

Infatti, possiamo avere problemi di:

- compatibilità dei tipi
- difetti nei domini
- rappresentazione dei dati
- . . .

Alcune strategie sono:

- approccio **top-down**: si testano prima i metodi che richiamano altri metodi
  - si sviluppano **stub**
- approccio **bottom-up**: si testano prima i metodi foglia (che non chiamano altri metodi)
  - si sviluppano **driver**
- **big bang**
  - uso contemporaneo di **top-down** e **bottom-up**

Regression Testing

> Si tratta del testing effettuato in fase di **manutenzione** del SW

H4

I suoi obiettivi sono:

- accertarsi di aver eliminato i bug segnalati
- accertarsi di non aver introdotto nuovi bug
- riusare i vecchi casi di test e accertarsi che per essi non vi siano malfunzionamenti, cioè che il loro comportamento rimanga corretto
- evitare di testare anche ciò che non è cambiato

H3 Aspetti

Gli **aspetti** da testare corrispondono alle proprietà del SW che abbiamo descritto qui.

H3 Accessibilità

A seconda del tipo "accesso" all'unità testata si ha:

- ***white box* testing**, o ***structural* testing**, o ancora ***program-based* testing**:
  si assume che *il sorgente sia disponibile*
- ***black box* testing**, o ***functional* testing**:
  si assume che *il sorgente **non** sia disponibile*, ma si analizzi solo quello che l'unità dovrebbe fare
- ***grey box* testing**:
  mix tra i due

White box testing

> È basato sulla **struttura interna** del programma.

H4

I casi di test sono derivati dal sorgente e durante l'esecuzione si analizza il comportamento del codice durante l'esecuzione. Ad esempio i *test di copertura* sono del tipo **white box**.

Passi principali

1. Esamina la struttura del programma
    - quali sono i punti critici, le decisioni principali, etc.

H5

2. Trova i casi di test che soddisfano un certo **criterio di copertura**
3. Manda in input tali casi di tesi uno alla volta e osserva il programma
4. Controlla che non vi siano malfunzionamenti

Vantaggi e svantaggi

✔ Il sorgente è un'importante fonte di informazione disponibile

✕ non riesce a trovare casi particolari

H5

✕ non fornisce ***test oracle***

> I **test oracle** sono meccanismi per stabilire se un test è passato o è fallito

Black box testing

> Ignora la struttura interna del programma e considera solo i **requisiti**

H4

I casi di test sono derivati dai **requisiti** e si analizza il programma solo attraverso la sua interfaccia esterna, cioè I/O.

Passi principali

1. Esamina la specifica dei requisiti del programma
2. Seleziona un insieme di casi di test che soddisfano qualche criterio

H5

3. Applica questi input alla specifica e colleziona gli **output attesi** (**A**)
4. Applica gli stessi input al programma e colleziona gli **output osservati** (**B**)
5. Confronta **A** e **B** e controlla che siano uguali

Vantaggi e svantaggi

✔ La specifica funziona da **oracolo**

✕ Le specifiche potrebbero non essere disponibili

H5

✕ Le specifiche devono essere **"formali"**, per generare i casi di test e poterli eseguire

✕ Maggiore sforzo rispetto al **white box**