Altri criteri

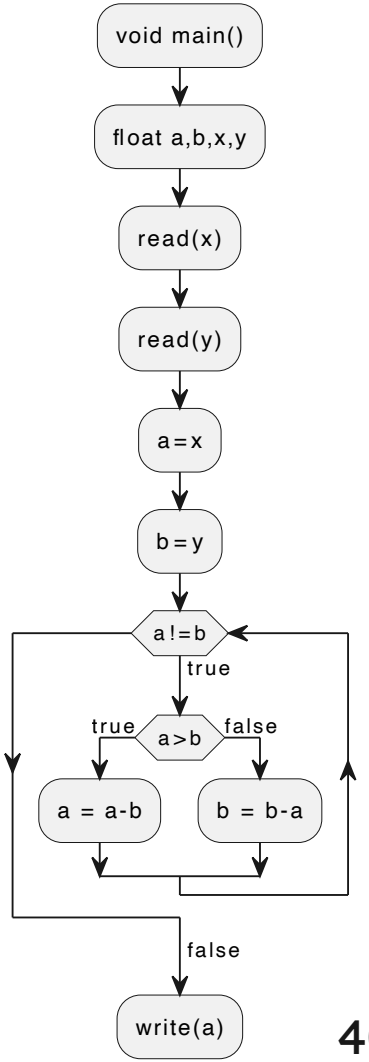
I criteri visti finora non considerano i **cicli** e possono essere soddisfatti da test che percorrono ogni ciclo al più una volta

Esperienza: molti errori si verificano durante iterazioni successive alla prima

- superamento limiti di un array

Occorre un criterio che tenga conto delle iterazioni

```
1  void main(){
2    float a,b,x,y;
3    read(x);
4    read(y);
5    a=x;
6    b=y;
7    while (a!=b)
8      if (a>b)
9        a = a-b;
10     else
11       b = b-a;
12   write(a);
13 }
```



Copertura dei cammini

Un test T soddisfa il criterio di copertura dei cammini se e solo se ogni cammino del grafo di controllo del programma viene percorso per almeno un caso di test in T

La metrica è la frazione dei cammini percorsi su quelli effettivamete percorribili

Molto generale, ma spesso impraticabile (anche per programmi semplici)

N-Copertura dei cicli

Un test soddisfa il criterio di n-copertura se e solo se ogni cammino del grafo contenente al massimo un numero di iterazioni di ogni ciclo non superiore a n viene percorso per almeno un caso di test

Si limita il numero massimo di percorrenze dei cicli

Quale è il valore ottimale di n?

- crescita molto rapida dei casi di test necessari al crescere di n

caso N = 2

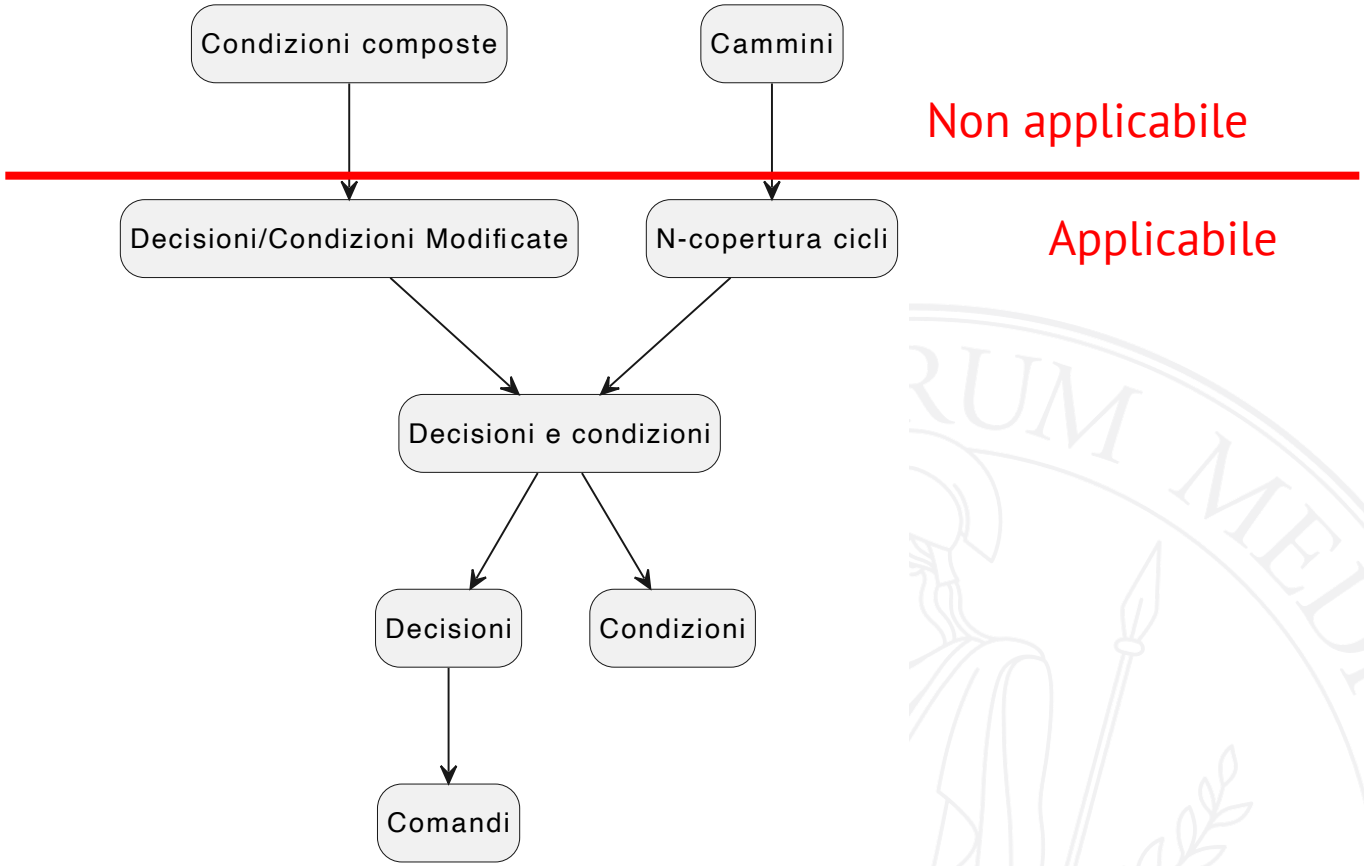
È il minimo per considerare cicli

se n = 1 un ciclo (`while`) sarebbe stato indistinguibile da una semplice selezione (`if`)

con n = 2 controllo

- casi in cui non devo entrare
- casi in cui entro una volta
- casi in cui rimango più di una volta

Implicazioni tra criteri di copertura aggiornato



Cosa si può ancora fare?

- Rimanendo nel testing strutturale possiamo fare una analisi più approfondita del codice
- Abbiamo considerato per adesso solo il flusso di controllo
- Possiamo fare considerazioni sul flusso dei dati?
  - necessita di una fase a priori di analisi statica

Analisi statica

- Si basa sull'esame di un insieme finito di elementi (le istruzioni del programma) contrariamente all'analisi dinamica (insieme degli stati delle esecuzioni)
- meno "costosa" del testing
  - non soffre del problema della "esplosione dello spazio degli stati"
- non può rilevare anomalie dipendenti da uno specifico valore assunto a run-time dalle variabili

Compilatori

- **analisi lessicale**: identificazione token del linguaggio, permette di identificare la presenza di simboli non appartenenti al linguaggio
- **analisi sintattica**: identifica relazione tra token; controlla conformità del codice alla grammatica del linguaggio
- **controllo dei tipi**: violazioni regole uso tipi
- **analisi flusso dei dati**: rleva problemi relativi a evoluzione del valore associato alle variabili

Analisi Data Flow

- I primi utilizzi sono stati fatti nel campo dell'ottimizzazione dei compilatori
- Il flusso dei dati sarebbe una analisi prettamente dinamica ma il sottoinsieme dei controlli statici è significativo
- Staticamente è possibile identificare il *tipo* di operazione che un comando esegue su una variabile
  - **definizione** se il comando assegna un valore alla variabile
  - **uso** se il comando richiede il valore di una variabile
  - **annullamento** se al termine dell'esecuzione dell'istruzione il valore della variabile non è significativo/affidabile

Analisi DF

Dal punto di vista di ciascuna variabile è possibile ridurre una sequenza di istruzioni (un cammino) ad una sequenza di

- **d** efinizioni
- **u** si
- **a** nnulamenti

Es. DF

```
1  void swap (int &x1, int &x2) {
2     int x;
3     x2 = x;
4     x2 = x1;
5     x1 = x;
6  }
```

Costruiamo le sequenze per ogni variabile:

x	auua
x1	…dud…
x2	…ddd…

Anomalie rilevabili:

- x viene usata (2 volte) senza essere stata prima definita
- x1 OK?
- x2 viene definita più volte senza essere usata nel frattempo

Regole DF

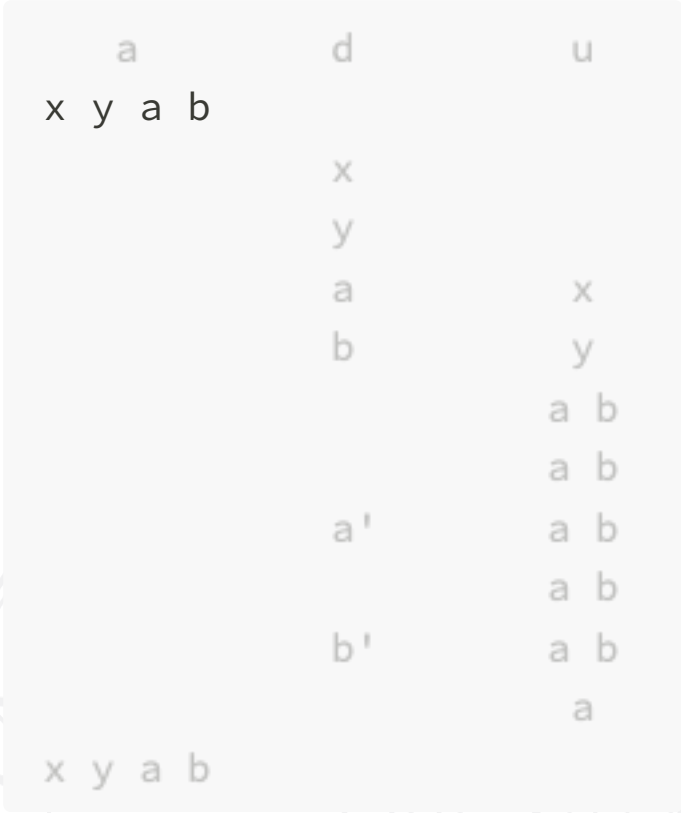
- L'*uso* di una variabile deve sempre essere preceduto in ogni sequenza da una definizione senza annullamenti intermedi
- La *definizione* di una variabile deve sempre essere seguita da un uso prima di un suo annullamento o definizione
- L'*annullamento* di una variabile deve essere sempre seguito da una definizione prima di un uso o di altro annullamento

	a	d	u
a	ERR		ERR
d	ERR	ERR	
u			

Non sono regole assolute: dipende dal linguaggio classificarle come *warning* o *error*

Esempio

```
1  void main(){
2    float a,b,x,y;
3    read(x);
4    read(y);
5    a=x;
6    b=y;
7    while (a!=b)
8      if (a>b)
9        a = a-b;
10     else
11       b = b-a;
12   write(a);
13 }
```



Sequenze DF

```
1  void main(){
2    float a,b,x,y;
3    read(x);
4    read(y);
5    a=x;
6    b=y;
7    while (a!=b)
8      if (a>b)
9        a = a-b;
10     else
11       b = b-a;
12   write(a);
13 }
```

$P(p, a)$ indica la sequenza ottenuta per la variabile a eseguendo il cammino p.

Es.
$P([1, 2, 3, 4, 5, 6, 7, 8, 9, 7, 12, 13], a) =$

$$a_2\, d_5\, u_7\, u_8\, u_9 d_9\, u_7\, u_{12}\, a_{13}$$

Per rappresentare P in caso di cammini contenenti cicli e decisioni si possono usare *espressioni regolari*

Esempio espressione regolare

```
1  void main(){
2    float a,b,x,y;
3    read(x);
4    read(y);
5    a=x;
6    b=y;
7    while (a!=b)
8      if (a>b)
9        a = a-b;
10     else
11       b = b-a;
12   write(a);
13 }
```

$P([1 \rightarrow], a)$

$a_2 \quad d_5$

$a_2 \quad d_5 \quad u_7(\quad \text{while-body} \quad)^* \quad u_{12} \quad a_{13}$

$a_2 \quad d_5 \quad u_7(\quad u_8 \quad \text{if-body} \quad)^* \quad u_{12} \quad a_{13}$

$a_2 \quad d_5 \quad u_7(\quad u_8 \quad (u_9 d_9 \mid u_{11}) \quad)^* \quad u_{12} \quad a_{13}$

$a_2 \quad d_5 \quad u_7(\quad u_8 \quad (u_9 d_9 \mid u_{11}) \quad u_7 \quad)^* \quad u_{12} \quad a_{13}$

- Ci sono coppie sbagliate?
- Era l'unica possibile?

Torniamo al testing e ai criteri di copertura

Osservazioni

- perché si presenti un malfunzionamento dovuto a una anomalia in una *definizione*, deve essere *usato* il valore che è stato assegnato
- un ciclo dovrebbe essere ripetuto (di nuovo) se verrà usato un valore definito alla iterazione precedente

Idea

- basare la selezione dei casi di test sulle sequenze definizione-uso delle variabili

Definizioni

- $\mathrm{def}(i)$ è l'insieme delle variabili che sono definite in i
- $\mathrm{du}(x, i)$ è l'insieme dei nodi j tali che:
  - $x \in \mathrm{def}(i)$
  - x usato in j
  - esiste un cammino da i a j, libero da definizioni di x

Criterio copertura definizioni

Un test T soddisfa il criterio di copertura delle definizioni se e solo se per ogni nodo i e ogni variabile x appartenente a $\mathrm{def}(i)$ T include un caso di test che esegue un cammino libero da definizioni da i ad almeno uno degli elementi di $\mathrm{du}(x, i)$

$T \in C$ sse

$\forall i \in P \; \forall x \in \mathrm{def}(i) \; \exists j \in \mathrm{du}(x, i) \; \exists t \in T$ che esegue un cammino da i a j senza ulteriori definizioni di x

Esempio copertura definizioni

```
1  void main(){
2    float a,b,x,y;
3    read(x);
4    read(y);
5    a=x;
6    b=y;
7    while (a!=b)
8      if (a>b)
9        a = a-b;
10     else
11       b = b-a;
12   write(a);
13 }
```

Ad esempio considerando la variabile a

$\mathrm{du}(a, 5) = \{7, 8, 9, 11, 12\}$

$\mathrm{du}(a, 9) = \{7, 8, 9, 11, 12\}$

$d_5 u_7$ viene "gratis"

$d_9 u_7$ basta entrare una volta nel ciclo

$T = \{< 8, 4 >\}$

Criterio copertura degli usi

Un test T soddisfa il criterio di copertura degli usi se e solo se per ogni nodo i e ogni variabile x appartenente a $\mathrm{def}(i)$ **per ogni elemento** j di $\mathrm{du}(x, i)$ T include un caso di test che esegue un cammino libero da definizioni da i a j

$T \in C$ sse

$\forall i \in P \; \forall x \in \mathrm{def}(i) \; \forall j \in \mathrm{du}(x, i) \; \exists t \in T$ che esegue un cammino da i a j senza ulteriori definizioni di x

Include il precedente?

Esempio copertura usi

```
1  void main(){
2    float a,b,x,y;
3    read(x);
4    read(y);
5    a=x;
6    b=y;
7    while (a!=b)
8      if (a>b)
9        a = a-b;
10     else
11       b = b-a;
12   write(a);
13 }
```

Considerando ancora la variabile a

$\mathrm{du}(a,5) = \{7,8,9,11,12\}$ e $\mathrm{du}(a,9) = \{7,8,9,11,12\}$

$d_5 u_7 u_8 u_{11} u_7 u_{12}$	$\ldots d_9 u_7 u_8 u_9 \ldots$
$\ldots d_5 u_7 u_8 u_9 \ldots$	$\ldots d_9 u_7 u_8 u_{11}$
	$\ldots d_9 u_7 \ldots u_{12} \ldots$

almeno due iterazioni

ad esempio $T = \{< 4,8 >, < 12,8 >, < 12,4 >\}$

Meglio minimizzare casi di test o iterazioni per caso?

Criterio copertura cammini DU

Esistono diversi cammini che soddisfano il criterio precedente.

Questo criterio richiede che siano selezionati tutti.

$T \in C$ sse

$\forall i \in P \; \forall x \in \text{def}(i) \; \forall j \in \text{du}(x, i)$ per ogni cammino da i a j senza ulteriori definizioni di x $\exists t \in T$ che lo esegue

Oltre le variabili

Analisi del flusso si può fare anche con altri "oggetti"

- FILE
  - **a** pertura (specializzata in *per lettura* o *per scrittura*)
  - **c** hiusura
  - **l** ettura
  - **s** crittura

Quali regole?

- l (s, c) deve essere preceduta da a senza c intermedie
- a deve essere seguita da c prima di altra a
- legame tipo apertura ed operazioni…

Altri criteri

Copertura del budget

- Ci si ferma quando
  - sono finite le risorse (tempo, soldi, …)
- Nessuna informazione sull'efficacia del test
- Spesso unico criterio applicato!!

https://en.wikipedia.org/wiki/Bebugging

Altri criteri

BeBugging

- Vengono introdotti deliberatamente n errori dentro il codice prima di mandare il programma a chi lo deve testare
  - Incentivo psicologico per il team di testing
    - sa che ci sono gli errori… deve solo trovarli
  - Metrica: percentuale di errori trovati
- Cercando gli errori inseriti apposta… troverò anche quelli inseriti per sbaglio

Analisi mutazionale

Viene generato un insieme di programmi Π *simili* al programma P in esame.

Su di essi viene eseguito lo stesso test T previsto per il programma P

Cosa ci si aspetta?

- se P è corretto allora i programmi in Π devono essere sbagliati
- per almeno un caso di test devono quindi produrre un risultato diverso

> Un test T soddisfa il criterio di copertura dei mutanti se e solo se per ogni mutante $\pi \in \Pi$ esiste almeno un caso di test in T la cui esecuzione produca per π un risultato diverso da quello prodotto da P

La metrica è la frazione di mutanti riconosciuta come diversa da P sul totale di mutanti generati.

Aspetti da tenere in conto

- analisi delle classi e generazione dei mutanti
- selezione dei casi di test
- esecuzione dei test

Generazione mutanti

- Quanto differiscono da P?
- Quanti sono?

CASO IDEALE:

- differenze minime
- un mutante per ogni possibile difetto
  - virtualmente infiniti

È però facilmente automatizzabile

Operatori mutanti

- Sono funzioni che, dato P, generano uno o più mutanti
- I più semplici effettuano modifiche sintattiche che comportino modifiche semantiche
  - ma non errori sintattici bloccati in compilazione
- HOM (High Order Mutation)
  - non solo una modifica
  - a volte sono più difficili da identificare che le due modifiche prese singolarmente

Classi di operatori

Si distinguono rispetto all'oggetto su cui operano

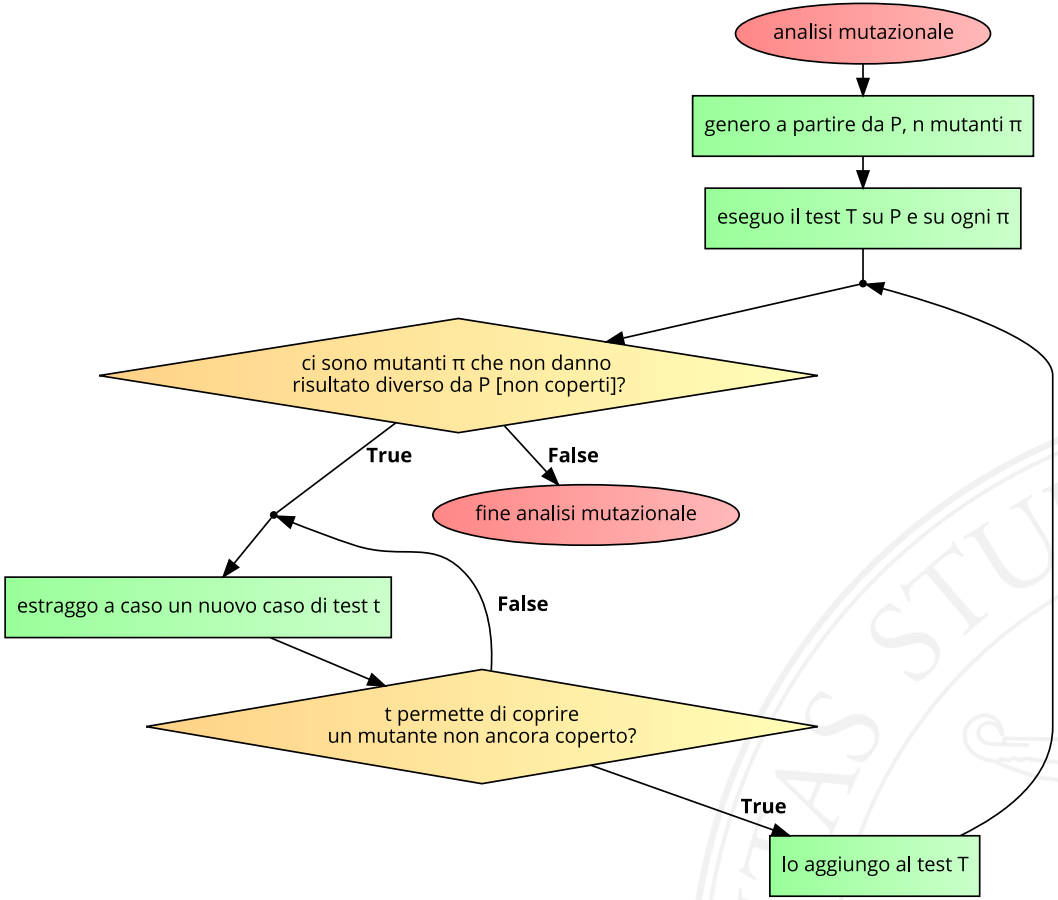
- costanti, variabili
  - es. scambiando l'occorrenza di una con l'altra
- operatori ed espressioni del programma
  - es. `<` in `<=`, le condizioni possono essere trasformate in `true` o `false`
- sui comandi del programma
  - es. un `while` in `if`

Possono essere specifici di alcuni tipi di applicazioni:

- Sistemi concorrenti
  - Operano principalmente sulle primitive di sincronizzazione
- Sistemi Object Oriented
  - Operano principalmente sulle interfacce dei moduli

Considerazioni

- Proliferare del numero di esecuzioni da effettuare per completare un test
  - abbiamo infatti che un caso di test non dà origine ad **una** esecuzione sola, ma ad $n + 1$ dove n è il numero di mutanti
- È possibile mediante opportuna infrastruttura automatizzare la generazione, l'esecuzione ed il controllo


analisi mutazionale
genero a partire da P, n mutanti π
eseguo il test T su P e su ogni π
ci sono mutanti π che non danno risultato diverso da P [non coperti]?
True
False
fine analisi mutazionale
estraggo a caso un nuovo caso di test t
False
t permette di coprire un mutante non ancora coperto?
True
lo aggiungo al test T

Cosa cambia con object orientation?

- Nei linguaggi procedurali il programma è composto da funzioni e procedure che si chiamano a vicenda scambiandosi dati tramite i parametri
  - le variabili globali sono deprecate
- Nei linguaggi OO, gli oggetti hanno dei metodi ad essi collegati, ma anche uno stato
  - I metodi non possono essere sempre significativamente testati isolatamente
- Cosa è una unità testabile?
  - Dalla procedura ci si sposta alla classe

OO testing e ereditarietà

- Basta testare un metodo una volta?
- Quello stesso metodo viene ereditato da una sottoclasse e va ritestato nel nuovo contesto
- Si possono testare le classi astratte?
  - Problema simile a testare una classe prima che sia completa
  - Si possono prevedere delle dummy implementazioni dei metodi astratti

OO testing e collegamento dinamico

- Complica sicuramente la determinazione dei criteri di copertura ad esempio perché non si possono più stabilire staticamente tutti i cammini

Class testing

Isoliamo la classe

- Costruiamo classi stub per renderla eseguibile indipendentemente dal contesto
- Implementiamo eventuali metodi astratti (metodo stub)
- Aggiungiamo una funzione che permetta di estrarre ed esaminare lo stato dell'oggetto
  - Per bypassare incapsulamento
- Costruiamo una classe driver che permetta di istanziare oggetti, e chiamare i metodi secondo il criterio di copertura scelto
  - Quale?

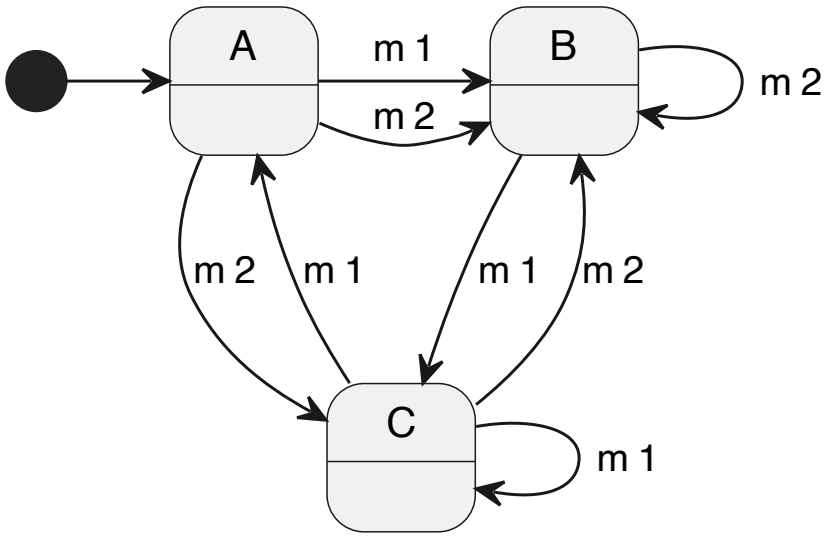
Copertura della classe

- Abbiamo detto che dobbiamo considerare lo stato dell'oggetto
- Abbiamo una definizione "statica" di cosa è lo stato dell'oggetto?
- Potrebbe esistere nella documentazione una rappresentazione come macchina a stati dell'oggetto che ci dice
  - gli stati
  - le transizioni (chiamate di metodi che cambiano lo stato)

Criteri di copertura

Abbiamo un diagramma… possiamo:

- Coprire tutti i nodi
  - Cioè coprire tutti gli stati del nostro oggetto
- Coprire tutti gli archi
  - Cioè coprire tutti i metodi per ogni stato
- Coprire tutti i cammini identificabili nel grafo
- Coprire tutte le coppie di archi in/out
  - Considero anche come sono arrivato in uno stato



Che tipo di test è?

white o black box?

- Abbiamo ipotizzato che esista questa rappresentazione… nelle specifiche
  - black box
- Se non esistesse potremmo ipotizzare di estrarre, mediante tecniche di reverse engineering, le informazioni sugli stati dal codice
  - white box

Test funzionale

- Non c'è (o meglio non si sfrutta) conoscenza del codice (*black box*)
- Può essere l'unico approccio possibile (test di validazione del lavoro di committente esterno)
- I dati di test possono essere derivati dalle specifiche (requisiti funzionali)
  - ci si concentra sul dominio delle informazioni invece che sulla struttura di controllo
- Permette di identificare errori non sintetizzabili con criteri strutturali
  - funzionalità non implementata
  - cammino di flusso dimenticato
- si pone come obiettivo anche trovare errori di interfaccia e di prestazioni

Tecniche

- Metodi basati su grafi
- Suddivisioni del dominio in classi di equivalenza
  - Category partition
- Analisi dei valori limite (test di frontiera)
- Collaudo per confronto

Testing delle interfacce

- Tipi di interfacce:
  - a invocazione con parametri
  - a condivisione di memoria
  - a metodi sincroni
  - a passaggio di messaggi
- Tipi di errori
  - sbaglio nell'uso dell'interfaccia
    - Ordine o tipo dei parametri
    - assunzioni sbagliate circa ciò che la funzione si attende
  - errori di tempistica o di sincronizzazione

Classi di equivalenza

- Si basa sulla suddivisione del dominio dei dati in ingresso in classi di dati, dalle quali derivare casi di test
  - una classe di dati è un insieme i cui componenti *dovrebbero* essere trattati in maniera analoga dal programma
- Si cerca quindi di individuare casi di test che rivelino (eventuali) classi di errori
  - es. elaborazione scorretta per numeri negativi
  - o per numeri molto grandi

Classi di equivalenza

Una classe di equivalenza rappresenta un insieme di stati validi o non validi per i dati in input e un insieme di stati validi per i dati di output

Un dato può essere ad esempio:

- un valore
- un intervallo
- un insieme di valori correlati

Suddivisione in classi

- Se ci si aspetta un valore specifico vengono definite una classe valida ed una non valida. es. un **codice PIN**:

I	PIN corretto
II	numero di 4 cifre qualsiasi != PIN

- Se ci si aspetta un valore in un intervallo, vengono definite una classe di equivalenza valida e due non valide. es. **[100,700]**:

I	numero tra 100 e 700
II	numero minore di 100
III	numero maggiore di 700

Test di frontiera

- Gli errori tendono ad accumularsi ai limiti del dominio
  - selezionare casi di test che esercitano i valori limite
- È complementare a classi di equivalenza:
  - non seleziono un elemento a caso della classe, ma uno ai confini

Category partition

È un particolare metodo di suddivisione in classi di equivalenza usabile a diversi livelli di granularità

- test di unità, di integrazione, di sistema

È composto dai seguenti passi:

1. Analizzare le specifiche
   - Identificare le *unità funzionali* individuali che possono essere verificate singolarmente.
   - Per ogni unità identificare le caratteristiche (*categorie*) dei parametri e dell'ambiente
2. scegliere dei valori (scelte) per le categorie
3. Determinare eventuali vincoli tra le scelte
4. (Scrivere test e documentazione)

Un esempio

Command

`find`

Syntax

$$find < pattern >< file >$$

Function

The find command is used to locate one or more instances of a given pattern in a file. All lines in the file that contain the pattern are written to standard output. A line containing the pattern is written only once, regardless of the number of times the pattern occur in it.

The pattern is any sequence of characters whose length does not exceed the maximum length of a line in the file. To include a blank in the pattern, the entire pattern must be enclosed in quotes ("). To include a quotation mark in the pattern, two quotes ("") in a row must be used.

Passo 1: analizzare le specifiche

Find è una funzione individuale che può essere verificata separatamente

Parametri: `pattern` **e** `file`

Caratteristiche parametro pattern

- *esplicite* (immediatamente derivabili dalle specifiche)
  - lunghezza del pattern
  - pattern tra doppi apici
  - pattern contenente spazi
  - pattern contenente apici
- *implicite* (nascoste nelle specifiche)
  - pattern tra apici con/senza spazi
  - più apici successivi inclusi nel pattern

file (parametro o ambiente?)

- **nome** file è un *parametro*
- **contenuto** file è *ambiente*

Caratteristiche parametro nomefile

- implicite
  - caratteri nel nome ammissibili o meno
  - file esistente (con permessi di lettura)

Caratteristiche ambiente file

- esplicite
  - numero occorrenze pattern nel file
  - max numero occorrenze pattern in una linea
  - max lunghezza linea
- implicite
  - pattern sovrapposti
  - tipo del file

Passo 2

Determinare i singoli casi significativi per ogni categoria identificata

È importante l'esperienza (know-how)

- vanno identificati tutti e soli i casi significativi

Paramentro pattern

Dimensione del pattern: vuoto, un solo carattere, più caratteri, più lungo di almeno una linea del file

Presenza di apici: pattern tra apici, pattern non tra apici, pattern tra apici errati

Presenza di spazi: nessuno, uno, molti

Presenza apici interni: nessuno, uno, molti

Passo 3

- Una combinazione di scelte per ogni parametro/ambiente rappresenta un caso di test plausibile (partizione)
- Generare un test per ogni combinazione delle classi fino qui identificate porta a 1.944 casi di test (troppi!)
- Come ridurli?

Approccio combinatorio

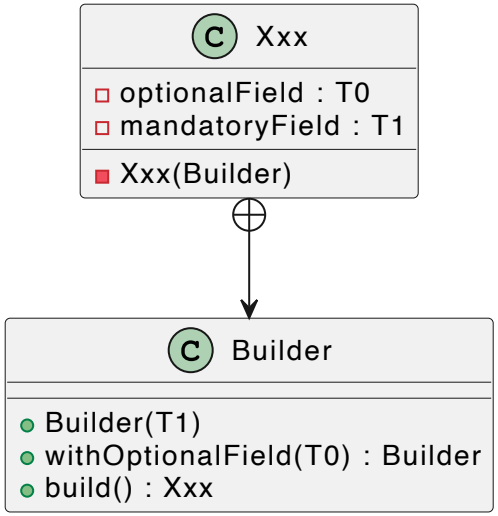
https://www.softwaretestinghelp.com/what-is-pairwise-testing/
https://www.pairwise.org/tools.html
https://pairwise.teremokgames.com/47wr0/

https://github.com/pavelicii/allpairs4j

```
public class Main {
 public static void main(String[] args) {
  AllPairs allPairs =
    new AllPairs.AllPairsBuilder()
        //.withTestCombinationSize(2)
        .withParameters(Arrays.asList(
            new Parameter("Category",
                "Sell", "Buy"),
            new Parameter("Location",
                "Mumbai", "Delhi"),
            new Parameter("Brand",
                "Mercedes", "BMW", "Audi"),
            new Parameter("Registration",
                "Valid", "Not Valid"),
            new Parameter("Order Type",
                "Ebooking", "InStore"),
            new Parameter("Order Time",
                "Working", "NotWorking")))
        .withConstraint(c ->
          c.get("Order Time").equals("NotWorking") &&
          c.get("Order Type").equals("InStore"))
        .printEachCaseDuringGeneration()
        .build();
  }
}
```

1. {Category=Sell, Location=Mumbai, Brand=Mercedes, Registration=Valid, Order Type=Ebooking, Order Time=Working}
2. {Category=Buy, Location=Delhi, Brand=BMW, Registration=Not Valid, Order Type=InStore, Order Time=Working}
3. {Category=Buy, Location=Mumbai, Brand=Audi, Registration=Not Valid, Order Type=Ebooking, Order Time=NotWorking}
4. {Category=Sell, Location=Delhi, Brand=Audi, Registration=Valid, Order Type=InStore, Order Time=Working}
5. {Category=Sell, Location=Delhi, Brand=BMW, Registration=Valid, Order Type=Ebooking, Order Time=NotWorking}
6. {Category=Buy, Location=Mumbai, Brand=Mercedes, Registration=Not Valid, Order Type=InStore, Order Time=Working}
7. {Category=Buy, Location=Mumbai, Brand=BMW, Registration=Valid, Order Type=InStore, Order Time=Working}
8. {Category=Sell, Location=Delhi, Brand=Mercedes, Registration=Not Valid, Order Type=InStore, Order Time=Working}
9. {Category=Sell, Location=Delhi, Brand=Mercedes, Registration=Not Valid, Order Type=Ebooking, Order Time=NotWorking}

BUILDER pattern




```java
public class Xxx {
   private final T1 mandatoryField;
   private final T0 optionalField1;
   private final T2 optionalField2;
   private Xxx(Builder builder) {
      mandatoryField = builder.mandatoryField;
      optionalField1 = builder.optionalField1;
      optionalField2 = builder.optionalField2;
   }
   public static class Builder {
      private T1 mandatoryField;
      private T0 optionalField1 = defaultValue1;
      private T2 optionalField2 = defaultValue2;
      public Builder(T1 mf) { mandatoryField = mf; }
      public Builder withOptionalField1(T0 of) {
         optionalField1 = of;
         return this;
      }
      public Builder withOptionalField2(T2 of) {
         optionalField2 = of;
         return this;
      }
      public Xxx build() { return new Xxx(this); }
   }
}
```

Builder pattern alternative

Telescoping constructor pattern

```
public class MyClass {
   private final T0 optionalField1;
   private final T1 mandatoryField;
   private final T2 optionalField2;

   public MyClass(T1 mf) {
      this(defaultValue1, mf, defaultValue2);
   }
   public MyClass(T1 mf, T0 of) {
      this(of, mf, defaultValue2);
   }
   public MyClass(T1 mf, T2 of) {
      this(defaultValue1, mf, of);
   }
   public MyClass(T1 mf, T0 of1, T2 of2) {
      ...
   }
}
```

JavaBeans pattern

```
public class MyClass {
   private T0 optionalField1;
   private T1 mandatoryField;
   private T2 optionalField2;

   public MyClass(T1 mf) {
      ...
   }
   public void setOptionalField1(T0 of) {
      optionalField1 = of;
   }
   public void setOptionalField2(T2 of) {
      optionalField2 = of;
   }
}
```

determinare vincoli

Più combinazioni di alcune scelte particolari possono essere poco significative:

- determinare vincoli per eliminare combinazioni meno significative
  - *proprietà* (ad esempio *NotEmpty* o *Quoted*)
  - *se* (limita l'uso di un valore solo ai casi in cui è definita una proprietà)
  - *single*
    - file che non contiene occorrenze del pattern cercato (che produce sempre lo stesso risultato indipendentemente dal tipo di pattern cercato)
  - *error*
    - nome del file omesso (che corrisponde ad un errore)

Passo 4

È possibile stimare il numero di casi di test e generare specifiche di test in modo automatico a partire da scelte e vincoli

Numero e distribuzione di vincoli possono essere determinati iterativamente stimando i casi di test

- generando le specifiche a partire da scelte e vincoli quando la stima dei casi di test è ragionevole
- valutando i test generati ed introducendo nuovi vincoli se in presenza di forti ridondanze

Testing funzionale e OO ?

Visto che non si basa sul codice …

- Problemi nella fase di integrazione
  - in generale non esiste la struttura gerarchica che possa guidare l'integrazione delle unità
- È possibile però trovare dei cluster significativi
  - Use cases e scenari
  - Sequence Diagram e copertura dei thread dei messaggi
  - State Diagram

Software inspection

- Tecnica manuale per individuare e correggere gli errori basata su di una attività di gruppo
- Fagan Code Inspections
  - la più diffusa tra le tecniche di ispezione (più rigorosa e definita)
  - estesa a fase di progetto e raccolta requisiti

Ruoli

Moderatore:

- in genere preso a prestito da un altro processo, coordina i meeting, sceglie i partecipanti, controlla il processo **Readers, Testers**:
- leggono il codice al gruppo, cercano difetti **Autore**:
- partecipante passivo; risponde ad eventuali domande

Software Inspection Process

- Planning
  - Il moderatore sceglie i partecipanti e fissa gli incontri
- Overview
  - per fornire il background e assegnare i ruoli
- Preparation
  - Attività svolte offline per la comprensione del codice o della struttura del sistema
- Inspection
- Rework
  - l'autore si occupa dei difetti individuati
- Follow-up:
  - possibile re-ispezione

Ispezione

Goal: trovare e registrare il maggior numero di difetti, ma non correggerli

- al massimo 2 sessioni di 2 ore al giorno approx. 150 source lines all'ora

Approccio: parafrasare linea per linea

- risalire allo scopo del codice a partire dal sorgente
- possibile anche "test a mano" trovare e registrare difetti, ma non correggerli
- checklist

From “Software Formal Inspections Guidebook", Office of Safety and Mission Assurance, NASA-GB-A302 August 1993

Checklist - esempio NASA

- Circa 2.5 pagine per il C, 4 per FORTRAN
  - Divise in: Functionality, Data Usage, Control, Linkage, Computation, Maintenance, Clarity
- Esempio:
  - Does each module have a single function?
  - Does the code match the Detailed Design?
  - Are all constant names upper case?
  - Are pointers not typecast (except assignment of NULL)?
  - Are nested “#include” files avoided?
  - Are non-standard usage isolated in subroutines and well documented?
  - Are there sufficient comments to understand the code?

Incentive structure

- Difetti trovati non devono essere utilizzati per la valutazione del personale
  - Il programmatore non va incentivato a nascondere difetti
- Difetti trovati dal test (dopo l'ispezione) sono usati per la valutazione del personale
  - Il programmatore è incentivato a trovare tutti i difetti durante l'ispezione

D. Parnas & D. Weiss, “Active design reviews: Principles and practices.” Proc. ICSE , Aug. 1985, pp. 132–136.

Variante: Active Design Reviews

- Osservazione:
  - Un revisore non preparato può stare seduto tranquillamente e non dire niente
- Variante al processo:
  - Scegliere revisori con adeguata esperienza
    - Diversi revisori per diversi aspetti
  - L’autore fa domande al revisore (checklist)
    - Il revisore dovendo rispondere è costretto a partecipare

Automazione dell'ispezione

Sebbene sia una tecnica manuale esistono tool di supporto:

- Controlli banali (e.g., formattazione)
- Riferimenti: Checklists, Standard con esempi
- Aiuti alla comprensione del codice
  - Tool comuni a quelli di attività di reengineering
    - Evidenziazione di parti rilevanti
    - Navigazione nel codice
    - Diversi tipi di rappresentazione dei dati e delle architetture
- Annotazioni & comunicazioni
- Guida al processo e rinforzo
  - non permettere di chiudere il processo se non sono soddisfatti alcuni

Funziona ?

La pratica ci dice che è *cost-effective* Perché?

- Processo rigoroso e dettagliato
- Basato su accumulo di esperienza (es. Checklist) si auto migliora
- Aspetti sociali del processo (riguardo all'autore soprattutto)
- Si considera l'intero dominio dei dati
- È applicabile anche a programmi incompleti

Limiti

- Livello del test: solo a livello di unità
- Non incrementale: evoluzione del software?

Fagan's Law (L17)
Inspections significantly increase productivity, quality, and project stability

https://web.archive.org/web/20060920113729/http://www2.umassd.edu/SWPI/ISERN/ISERN-98-10.pdf

Confronto tra le varie tecniche

Authors	Average Percentage of defects found by subjects applying		
	Inspection (Code Reading)	Functional Testing	Structural Testing
Hetzel [12]	37.3	47.7	46.7
Myers [21]	30.0	36.0	38.0
Basili & Selby [3]	54.1	54.6	41.2
Kamsties & Lott [17] (1. Replication)	40.0	37.0	36.0
Kamsties & Lott [17] (2. Replication)	50.3	53.4	33.5
Wood et al. [28]	43.41	55.05	57.87

https://web.archive.org/web/20070221162909/http://www2.umassd.edu/SWPI/TechnicalReview/r4094.pdf

… e metterle insieme ?

Table 2. Defect-removal efficiency under 16 permutations of four factors.

	Factors used				Results		
	Formal design inspection	Formal code inspection	Formal quality assurance	Formal testing	Worst	Median	Best
1		(None used)			30%	40%	50%
2			✓		32%	45%	55%
3				✓	37%	53%	60%
4		✓			43%	57%	66%
5	✓				45%	60%	68%
6			✓	✓	50%	65%	75%
7		✓	✓		53%	68%	78%
8		✓		✓	55%	70%	80%
9	✓		✓		60%	75%	85%
10	✓			✓	65%	80%	87%
11	✓	✓			70%	85%	90%
12		✓	✓	✓	75%	87%	93%
13	✓		✓	✓	77%	90%	95%
14	✓	✓	✓		83%	95%	97%
15	✓	✓		✓	85%	97%	99%
16	✓	✓	✓	✓	95%	99%	99.99%

Hetzel-Myers's Law (L20)
A combination of different V&V methods outperforms any single method alone

Vantaggi gruppo di test autonomo

Weinberg's Law (L23)
A developer is unsuited to test his or her code

- Aspetti tecnici
  - maggiore specializzazione
  - conoscenza delle tecniche e degli strumenti
- Aspetti psicologici
  - distacco dal codice
    - test indipendente da conoscenza codice
    - attenzione ad aspetti dimenticati
  - indipendenza della valutazione

https://web.archive.org/web/20220526084408/http://testobsessed.com/wp-content/uploads/2011/04/btwq.pdf

Svantaggi gruppo di test autonomo

- Aspetti tecnici
  - progressiva perdita di capacità di progetto e codifica
  - minore conoscenza dei requisiti
- Aspetti psicologici
  - possibile pressione negativa su team sviluppo
  - possibile gestione scorretta delle responsabilità

Possibili alternative

- Rotazione del personale
  - permette di evitare progressivo depauperamento tecnico dovuto a eccessiva specializzazione
  - permette di evitare svuotamento dei ruoli
  - aumenta i costi di formazione
  - aumenta le difficoltà di pianificazione
- Condivisione del personale
  - permette di supplire a scarsa conoscenza del prodotto in esame
  - aumenta le difficoltà di gestione dei ruoli

Modelli statistici

- relazione statistica tra metriche e
  - presenza errori (per classi di errori)
  - numero errori (per classi di errori)

Possibile predire distribuzione errori per modulo

Pareto-Zipf-type Laws (L24)
Approximately 80 percent of defects come from 20 percent of modules

Debugging

- Mira a localizzare e rimuovere le anomalie che sono le cause di malfunzionamenti riscontrati nel programma
- Non deve essere usato per rilevare malfunzionamenti
- L'attività è definita per un programma e un insieme di dati che causano malfunzionamenti nel programma
  - Si basa su riproducibilità del malfunzionamento
  - Va verificato che il "malfunzionamento" non sia dovuto a specifiche errate

> *"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."*
>
> — Brian W. Kernighan

Problemi

- Non *sempre* facile stabilire una relazione anomalia-malfunzionamento
- Non esiste una relazione biunivoca tra anomalie e malfunzionamento

10 little Indian boys went out to dine;
One choked his little self and then there were 9.

9 little Indian boys sat up very late;
One overslept himself and then there were 8.

10 little bugs were in the code.
Take one down, patch it around.

27 little bugs were in the code…

Tecnica Naïve

Consiste nell'introdurre nel modulo in esame comandi di uscita che stampino il valore intermedio assunto dalle variabili

- facile da applicare (bastano un compilatore e un esecutore)
- richiede la modifica del codice (e quindi la sua rimodifica una volta individuata la anomalia)
- poco flessibile (modifica e compilazione per ogni nuovo stato)

Tecnica Naïve “avanzata”

Un miglioramento parziale si può ottenere sfruttando funzionalità del linguaggio

- ad esempio in C #ifdef e -D
- librerie di logging (con tipologia messaggi)
- asserzioni
  - possono essere viste anche come oracoli interni al codice

Dump di memoria

Consiste nel produrre una immagine esatta della memoria dopo passo di esecuzione

- non richiede modifica del codice
- spesso difficile per la differenza tra la rappresentazione astratta dello stato (legata alle strutture dati del linguaggio utilizzato) e la rappresentazione fornita dallo strumento
- viene prodotta una mole di dati per la maggior parte inutile

Debugging simbolico

Gli stadi intermedi sono prodotti usando una rappresentazione compatibile con quella del linguaggio usato

- Gli stati sono rappresentati come strutture dati e valori ad esse associati
- I debugger simbolici forniscono ulteriori strumenti (watch o spy monitor) che permettono di visualizzare il comportamento del programma in maniera selettiva
  - inserimento breakpoint, watch su variabili

Debugging per prova

Non solo la visualizzazione ma anche l'esame automatico degli stati ottenuti

- strumenti per verificare la "correttezza" degli stadi intermedi
- watch condizionali
- asserzioni a livello di monitor (invece che nel codice)
  - ad esempio si chiede al monitor di controllare che gli indici di un array siano sempre interni all'intervallo

Altre funzionalità debugger

- Gestire la granularità del passo di esecuzione:
  - singolo passo
  - entrare dentro a funzione
  - drop/reset del frame
- Modificare il contenuto di una variabile (o zona di memoria)
- Modificare codice
  - non sempre possibile… necessita ricompilazione ma poi si prosegue dal punto in cui ci si era interrotti
- Rappresentazioni grafiche dei dati

Visualizzazione degli stream

```
public static void main(String[] args) {
  Day01 d = new Day01();

  System.out.println(d.inputAsScanner()
      .useDelimiter("\n\n").tokens()
      .map(s -> new Scanner(s)
          .tokens()
          .mapToLong(Long::parseLong)
          .sum())
      .sorted(Comparator.reverseOrder())
      .limit(3)
      .mapToLong(Long::valueOf)
      .sum()
  );
}
```

È possibile automatizzare il debugging?

http://www.st.cs.uni-saarland.de/dd/
https://www.debuggingbook.org/html/DeltaDebugger.html

Delta (differential) debugging

- Andreas Zeller (Saarlamd University, Saarbrucken, Germany)