
UNIVERSITÀ DEGLI STUDI
DI MILANO

# Ingegneria del software

## Progettazione


UNIVERSITÀ DEGLI STUDI
DI MILANO

# Verifica e Convalida

# Codice Lab 08

- correzione
  - caso con filtro ma set lettere vuoto
  - mancato settaggio FormatStrategy di default
- refactoring
  - pattern null per le strategie
  - pattern builder (Item 2 di Effective Java)



# Builder pattern alternative

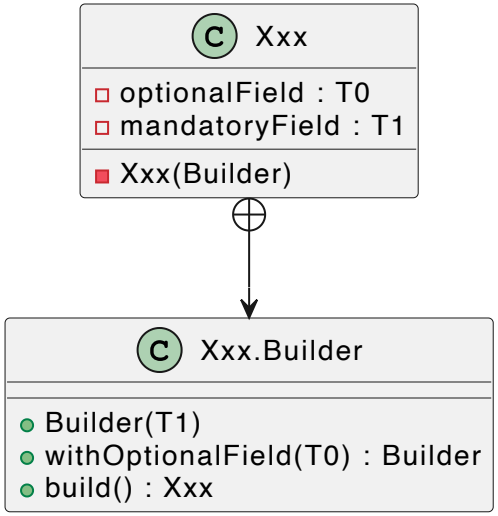
## Telescoping constructor pattern

```
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;
   }
}
```

# 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); }
   }
}
```

# Altri criteri

## Beebugging

- 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 $\Pi$ *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 $\Pi$ 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 $\pi$ 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

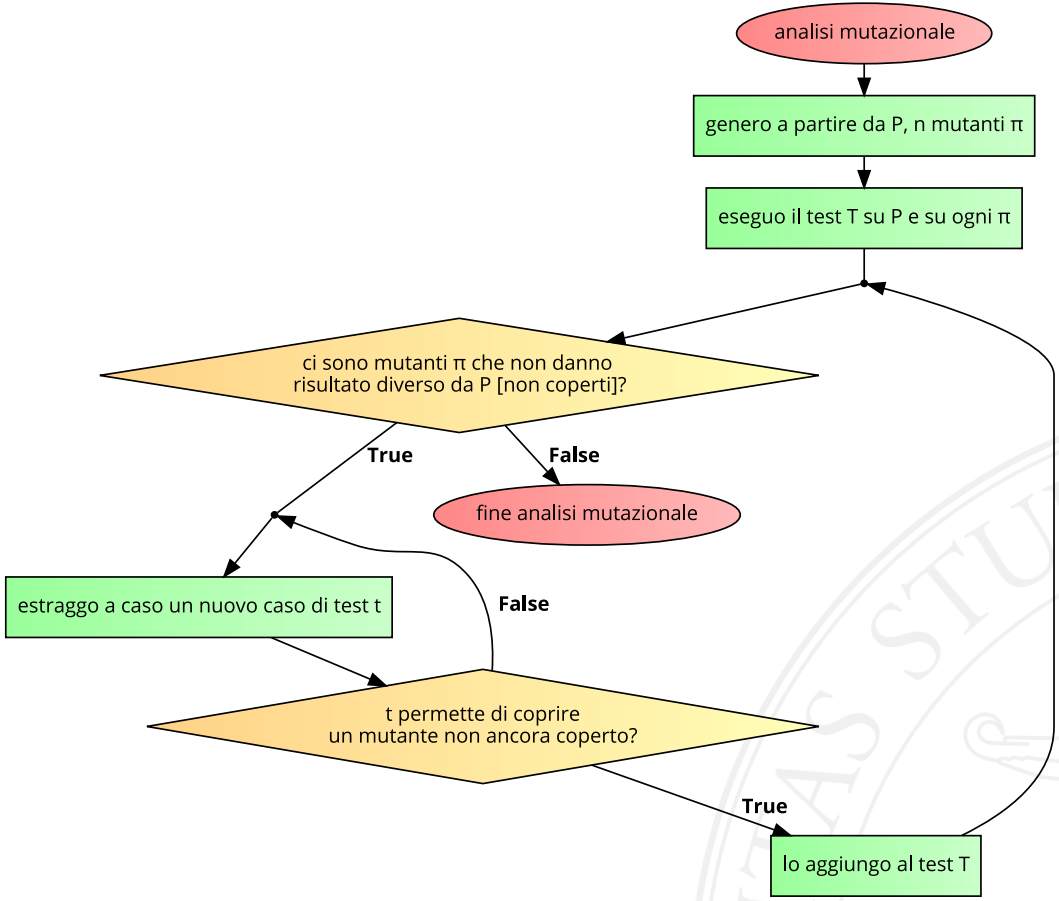
- costanti, variabili
  - es. scambiando l'occorrenza di una con l'altra
- operatori ed espressioni del programma
  - es. `<` in `<=`, le condizioni possono essere trasformate in `true` o `false`
- sui comandi del programma
  - es. un `while` in `if`

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?

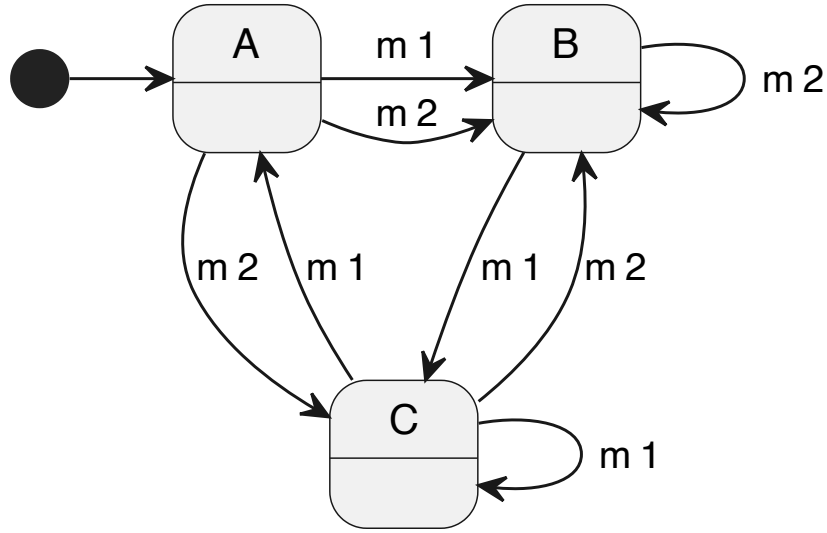
# Copertura della classe

- 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 tutte le coppie di archi in/out
  - Considero anche come sono arrivato in uno stato
- Coprire tutti i cammini identificabili nel grafo



# 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