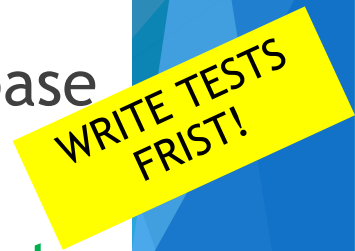
IL PRINCIPIO DI SOSTITUZIONE DI LISKOV

Fasi della progettazione e dello sviluppo

Idealmente, la progettazione e lo sviluppo di un programma Java dovrebbe attraversare le seguenti fasi:

1. Definizione della gerarchia di classi e interfacce da realizzare
2. Identificazione dei membri pubblici di ogni classe
3. Definizione delle SPECIFICHE di ogni metodo pubblico (condizioni sui parametri, sul risultato e comportamento atteso del metodo)
4. Implementazione di programmi di testing delle singole classi sulla base di quanto definito (membri pubblici e specifiche)
5. Definizione dei membri privati seguendo il principio di incapsulamento
6. Implementazione del codice delle classi, da verificare con i test già sviluppati



Definire le specifiche

Definire le specifiche di una classe significa esprimere in modo non ambiguo il comportamento atteso dei suoi metodi pubblici

Come fare:

- Aggiungendo opportuni commenti al codice
- Esprimendo delle condizioni sui parametri e sulle variabili che siano verificabili o testabili, ad esempio, con `assert`

Specifiche: esempio `IntSet`

Esempio di specifica con commenti:

```
// OVERVIEW: un IntSet è un insieme di interi
public class IntSet {

    public IntSet(int capacity) {
    // REQUIRES: capacity non negativo
    // EFFECTS: crea un insieme vuoto che può ospitare capacity elementi
    }

    public boolean add(int elem) throws FullSetException {
    // REQUIRES: numero di elementi contenuti nell'insieme minore di capienza
    // EFFECTS: se elem non è presente nell'insieme lo aggiunge e restituisce true,
    //          restituisce false altrimenti
    }

    public boolean contains(int elem) {
    // REQUIRES:
    // EFFECTS: restituisce true se elem p presente nell'insieme, false altrimenti
    }
}
```

Specifiche: esempio `IntSet`

Esempio di specifica con commenti:

```
// OVERVIEW: un IntSet è un insieme di interi
public class IntSet {

    public IntSet(int capacity) {
    // REQUIRES: capacity non negativo
    // EFFECTS: crea un insieme vuoto che può ospitare
    }

    public boolean add(int elem) throws FullSetExceptio
    // REQUIRES: numero di elementi contenuti nell'insi
    // EFFECTS: se elem non è presente nell'insieme lo aggiunge e restituisce true,
    //          restituisce false altrimenti
    }

    public boolean contains(int elem) {
    // REQUIRES:
    // EFFECTS: restituisce true se elem p presente nell'insieme, false altrimenti
    }
}
```

Può sollevare un'eccezione:

- funzionamento delle eccezioni con trow e try/catch simile a JavaScript
- Le eccezioni in Java possono essere checked (estendono Exception) o unchecked (estendono RuntimeException)
- I metodi che sollevano eccezioni checked lo devono dichiarare nell'intestazione (con throws) e il chiamante è obbligato (dal compilatore) a controllarle con try/catch

Specifiche: esempio `IntSet`

Java mette a disposizione anche una sintassi specifica per le specifiche tramite commenti (JavaDoc)

```
...

    /**
     * Aggiunge un elemento all'insieme
     * @param elem valore intero
     * @return true se l'inserimento viene aggiunto, false se già presente
     * @throws FullSetException se l'insieme è pieno
     */
    public boolean add(int elem) throws FullSetException {
        ...
    }

...
```

Specifiche: esempio `IntSet`

Java
spec

Da questi commenti si può generare una documentazione della classe in formato HTML tramite il tool JavaDoc (incluso nel JDK)

```
javadoc IntSet.java
```

...

add

```
public boolean add(int elem)
            throws FullSetException
```

Aggiunge un elemento all'insieme

Parameters:

`elem` - valore intero

Returns:

true se l'inserimento viene aggiunto, false se gia' presente

Throws:

`FullSetException` - se l'insieme e' pieno

...

nte

Precondizioni e postcondizioni

Per ogni metodo, le specifiche indicano:

- PRECONDIZIONI (//REQUIRES) : condizioni sui parametri e sul valore delle variabili che devono essere soddisfatte all'inizio dell'esecuzione del metodo
- POSTCONDIZIONI (//EFFECTS) : condizioni sul risultato e sul valore delle variabili che devono essere soddisfatte al termine dell'esecuzione del metodo, assumendo che all'inizio fossero vere le precondizioni

Precondizioni e postcondizioni

```
...

    public boolean add(int elem) throws FullSetException {
    // REQUIRES: numero di elementi contenuti nell'insieme minore di capienza
    // EFFECTS: se elem non è presente nell'insieme lo aggiunge e restituisce true,
    //          restituisce false altrimenti
    }

...
```

Se all'inizio è vera la precondizione

numero elementi < capienza

alla fine deve essere vera la postcondizione

risultato == (elem ∈ insieme)

Dalla specifica al tester

Sulla base della sola specifica possiamo già scrivere un tester della classe IntSet

Collauda la classe controllando se i metodi implementati rispettano la relazione tra precondizioni e postcondizioni sugli argomenti più frequenti e su quelli che rappresentano casi limite (es. size=0)

```
public class Tester {

    public static void main (String[] args) {

        IntSet is = new IntSet(10);

        boolean t1 = is.add(5); // TEST 1
        if (t1) System.out.println("TEST 1 [add 5] SUPERATO");
        else System.out.println("TEST 1 [add 5] FALLITO");

        boolean t2 = is.contains(5); // TEST 2
        if (t2) System.out.println("TEST 2 [contains 5] SUPERATO");
        else System.out.println(" TEST 2 [contains 5] FALLITO");

        ...

    }
}
```

Dalla specifica all'implementazione

Ecco un'implementazione di IntSet che soddisfa le specifiche:

```
// OVERVIEW: un IntSet è un insieme di interi
public class IntSet {

   protected int[] a;  // RAPPRESENTAZIONE: ARRAY DI INTERI + SIZE
   protected int size;

   public IntSet(int capacity) {
   // REQUIRES: capacity non negativo
   // EFFECTS: crea un insieme vuoto che può ospitare capacity elementi
       a = new int[capacity]; // SE capacity<0, SOLLEVA NegativeArraySizeException
         this.size = 0;
   }

... (SEGUE)
```

Dalla specifica all'implementazione

Ecco un'implementazione di IntSet che soddisfa le specifiche:

```
... (SEGUE)

    public boolean add(int elem) throws FullSetException {
    // REQUIRES: numero di elementi contenuti nell'insieme minore di capienza
    // EFFECTS: se elem non è presente nell'insieme lo aggiunge e restituisce true,
    //          restituisce false altrimenti
        if (size>=a.length) throw new FullSetException();
        else if (contains(elem)) return false;
         else { a[size]=elem; size++; return true; }
    }

    public boolean contains(int elem) {
    // REQUIRES:
    // EFFECTS: restituisce true se elem p presente nell'insieme, false altrimenti
         for (int i=0; i<size; i++) {
             if (a[i]==elem) return true;
         }
         return false;
    }
}
```

Verifica delle specifiche

I test consentono di identificare errori di programmazione (rispetto alle specifiche)

- Non sono esaustivi (spesso non possono testare tutti i casi possibili)
- Esistono strumenti di generazione automatica di test (dalle specifiche) che consentono di aumentare la copertura dei casi da provare

Esistono metodi di analisi statica e model checking che consentono di verificare (dimostrare) in modo automatico la correttezza del codice rispetto alle specifiche

- Necessari in casi di programmi eseguiti in contesti safety-critical e cybersecurity
- Esempi:
  - Infer (https://fbinfer.com/) analizzatore statico per Java/C/C++ (sviluppato da Facebook)
  - PathFinder (https://github.com/javapathfinder/jpf-core/wiki) model checker per Java (sviluppato alla Nasa)
- Metodi approfonditi in corsi della magistrale…

Principio di Sostituzione di Liskov
(Liskov Substitution Principle – LSP)

L'esistenza di una specifica consente di ragionare sul rapporto tra superclasse e sottoclasse da un punto di vista comportamentale

Principio di Sostituzione di Liskov (LSP)

Un oggetto di un sottotipo può sostituire un oggetto del supertipo senza influire sul comportamento dei programmi che usano il supertipo

La sottoclasse deve soddisfare le specifiche della superclasse

Principio di Sostituzione di Liskov
(Liskov Substitution Principle – LSP)

Esempio: FlexIntSet sottoclasse di IntSet per insiemi ridimensionabili

```
// OVERVIEW: un FlexIntSet è un insieme di interi ridimensionabile
public class FlexIntSet extends IntSet {

    // protected int[] a;  // EREDITATI (STESSA RAPPRESENTAZIONE)
    // protected int size;

    public FlexIntSet(int capacity) {
        super(capacity); // SE capacity<0, SOLLEVA NegativeArraySizeException
    }

... (SEGUE)
```

Principio di Sostituzione di Liskov

```
... (SEGUE)

    public boolean add(int elem) {
    // REQUIRES:
    // EFFECTS: se elem non è presente nell'insieme lo aggiunge e restituisce true,
    //          restituisce false altrimenti
        if (size>=a.length) {                    // se array pieno
            int[] tmp = new int[size+10];        // crea un nuovo array più grande
            for (int i=0; i<size; i++)
                tmp[i] = a[i];                   // copia gli elementi
            a = tmp;                             // aggiorna il riferimento
        }
        if (contains(elem)) return false;
        else { a[size]=elem; size++; return true; }
    }

    // public boolean contains(int elem)       EREDITATO

}
```

Principio di Sostituzione di Liskov
(Liskov Substitution Principle – LSP)

Il metodo della sottoclasse
- ha precondizioni più deboli
- non solleva eccezioni

Confrontiamo le specifiche del metodo add:

- Nella superclasse IntSet:

```
public boolean add(int elem) throws FullSetException {
// REQUIRES: numero di elementi contenuti nell'insieme minore di capienza
// EFFECTS: se elem non è presente nell'insieme lo aggiunge e restituisce true,
//          restituisce false altrimenti
```

- Nella sottoclasse FlexIntSet:

```
public boolean add(int elem) {
// REQUIRES:
// EFFECTS: se elem non è presente nell'insieme lo aggiunge e restituisce true,
//          restituisce false altrimenti
```

Principio di Sostituzione di Liskov
(Liskov Substitution Principle – LSP)

E' facile vedere che gli oggetti della classe FlexIntSet sono sostituibili (secondo la definizione di Liskov) agli oggetti della classe IntSet

- In qualunque contesto che faccia un uso appropriato di IntSet (i.e., che chiami i suoi metodi in modo appropriato rispetto alle precondizioni) si può usare un FlexIntSet ottenendo esattamente lo stesso comportamento

```
IntSet set = new IntSet(10);    // esempio di contesto in cui si usa IntSet
for (int i=0; i<10; i++)        // inserisce 1,1,2,2,3,3,4,4,5,5
    set.add(i/2);
System.out.println(set.contains(3));  // true
System.out.println(set.contains(6));  // false
```

Principio di Sostituzione di Liskov
(Liskov Substitution Principle – LSP)

E' facile vedere che gli oggetti della classe FlexIntSet sono sostituibili (secondo la definizione di L

- In qualunque contesto ch che chiami i suoi metodi precondizioni) si può usa stesso comportamento

Il comportamento del programma non cambia!

- questo non sarebbe vero se il contesto cercasse di inserire 11 elementi diversi, ma in quel caso starebbe violando le precondizioni di IntSet (e non quelle di FlexIntSet)

```
IntSet set = new FlexIntSet(10);   // sostituzione IntSet -> FlexIntSet
for (int i=0; i<10; i++)          // inserisce 1,1,2,2,3,3,4,4,5,5
    set.add(i/2);
System.out.println(set.contains(3));  // true
System.out.println(set.contains(6));  // false
```

Altro esempio: Rettangolo e Quadrato

Consideriamo una classe Rectangle con le seguenti specifiche

```
public class Rectangle {

    public void setWidth(int w) {
    // REQUIRES: w non negativo
    // EFFECTS: imposta la larghezza a w
    }
    public void setHeight(int h) {
    // REQUIRES: h non negativo
    // EFFECTS: imposta l'altezza ad h
    }
    public int area() {
    // EFFECTS: calcola larghezza*altezza
    }
}
```

Un quadrato è un caso particolare di rettangolo che ha larghezza sempre uguale ad altezza … sottoclasse!

Altro esempio: Rettangolo e Quadrato

```
public class Rectangle {

    private int width;
    private int height;

    public void setWidth(int w) {
        if (w>=0) width = w;
    }
    public void setHeight(int h) {
        if (h>=0) height = h;
    }

    public int area() {
        return width*height;
    }
}
```

```
public class Square extends Rectangle {

    @Override
    public void setWidth(int w) {
        // aggiorna entrambe le dimensioni
        if (w>=0) {width = w; height = w; }
    }

    @Override
    public void setHeight(int h) {
        // aggiorna entrambe le dimensioni
        if (h>=0) { height = h; width = h;}
    }

    // public int area()   EREDITATO

}
```

NOTA: @Override è una annotazione (facoltativa) che aumenta la leggibilità del codice e consente al compilatore di fare qualche controllo in più (ad es., che stiamo effettivamente facendo un overriding)

Altro esempio: Rettangolo e Quadrato

Esempio di contesto d'uso:

```
Rettangolo r = new Rettangolo();
r.setWidth(3);
r.setHeight(2);
System.out.println("AREA: " + r.area());   // AREA: 6
```

```
Rettangolo r = new Quadrato();
r.setWidth(3);
r.setHeight(2);
System.out.println("AREA: " + r.area());   // AREA: 4
```

Il comportamento è diverso!

- la sottoclasse Square, sebbene sia corretta dal punto di vista dei tipi, non soddisfa il principio di sostituzione di Liskov

Altro esempio: Rettangolo e Quadrato

Dove sta il problema?

- La sottoclasse Square non soddisfa le specifiche della superclasse Rectangle

```
public class Rectangle {

    public void setWidth(int w) {
    // REQUIRES: w non negativo
    // EFFECTS: imposta la larghezza a w
    }
    public void setHeight(int h) {
    // REQUIRES: h non negativo
    // EFFECTS: imposta l'altezza ad h
    }
    public int area() {
    // EFFECTS: calcola larghezza*altezza
    }
}
```

L'effetto di setWidth e setHeight di Square è di cambiare sia la larghezza che l'altezza

Altro esempio: Rettangolo e Quadrato

Possibile soluzione: interfaccia Shape

```
public interface Shape {
    public int area();
    // EFFECTS: calcola l'area della forma
}
```

```
public class Rectangle implements Shape {

    public void setWidth(int w) {
    // REQUIRES: w non negativo
    // EFFECTS: imposta la larghezza a w
    }
    public void setHeight(int h) {
    // REQUIRES: h non negativo
    // EFFECTS: imposta l'altezza ad h
    }
    public int area() {
    // EFFECTS: calcola larghezza*altezza
    }
}
```

```
public class Square implements Shape {
    // non serve l'altezza...
    public void setWidth(int w) {
    // REQUIRES: w non negativo
    // EFFECTS: imposta la larghezza a w
    }
    public int area() {
    // EFFECTS: calcola larghezza^2
    }
}
```

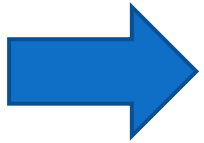
Principio di Sostituzione di Liskov
(Liskov Substitution Principle – LSP)

Il principio di sostituzione di Liskov va oltre il semplice concetto di sottotipo sostituibile per subsumption

- Esprime una relazione tra i comportamenti dei oggetti dei due tipi
- La relazione deve valere per tutti i possibili contesti in cui la sostituzione può avvenire

Problemi:

- I contesti possibili sono infiniti
- I contesti sono programmi



Verificare se il principio LSP vale tra due classi è un problema indecidibile

- il compilatore non può farlo…

Regole indotte dal LSP

Il principio di sostituzione di Liskov nella pratica si traduce in un insieme di regole da seguire:

- la regola della segnatura
  - gli oggetti del sotto-tipo devono avere tutti i metodi del super-tipo
  - le segnature (firme) dei metodi del sotto-tipo devono essere compatibili con le segnature dei corrispondenti metodi del super-tipo
- la regola dei metodi
  - le chiamate dei metodi del sotto-tipo devono comportarsi come le chiamate dei corrispondenti metodi del super-tipo
- la regola delle proprietà
  - il sotto-tipo deve preservare tutte le proprietà che possono essere provate sugli oggetti del super-tipo

Regola sui tipi

Regole semantiche (sul comportamento)

Regola della segnatura

- la regola della segnatura
  - gli oggetti del sotto-tipo devono avere tutti i metodi del super-tipo
  - le segnature (firme) dei metodi del sotto-tipo devono essere compatibili con le segnature dei corrispondenti metodi del super-tipo

La presenza di tutti i metodi è garantita dal compilatore Java tramite i meccanismi di ereditarietà (extends)

In caso di overriding, il metodo nella sottoclasse:

- deve avere la stessa firma del metodo nella superclasse
- può sollevare meno eccezioni
- può avere un return type più specifico (sottotipo) di quello del metodo della superclasse

se il tipo del metodo della super-classe è

$$S \rightarrow T_1$$

il tipo del metodo della sotto-classe può essere:

$$S \rightarrow T_2$$

con $T_2 <: T_1$ (covariante)

Regola dei metodi

- la regola dei metodi
  - le chiamate dei metodi del sotto-tipo devono comportarsi come le chiamate dei corrispondenti metodi del super-tipo

- In generale, un sotto-tipo può indebolire le precondizioni e rafforzare le postcondizioni dei metodi che sono riscritti
- Per avere compatibilità tra la specifica di un metodo **m** del super-tipo e la specifica del metodo **m** del sotto-tipo, devono essere soddisfatte le regole:
  - regola della pre-condizione $\boldsymbol{pre_{super} \Rightarrow pre_{sub}}$
  - regola della post-condizione $\boldsymbol{(pre_{super} \,\&\&\, post_{sub}) \Rightarrow post_{super}}$

Vediamo nel metodo add di IntSet/FlexIntSet:

- Nella superclasse IntSet:

```
public boolean add(int elem) throws FullSetException {
// REQUIRES: numero di elementi contenuti nell'insieme minore di capienza
// EFFECTS: se elem non è presente nell'insieme lo aggiunge e restituisce true,
//          restituisce false altrimenti
```

- Nella sottoclasse FlexIntSet:

```
public boolean add(int elem) {
// REQUIRES:
// EFFECTS: se elem non è presente nell'insieme lo aggiunge e restituisce true,
//          restituisce false altrimenti
```

Dalle specifiche (REQUIRES/EFFECTS):

pre_{super} = (size<capacity) $post_{super}$ = (retval==(elem ∈ set))

pre_{sub} = true $post_{sub}$ = (retval==(elem ∈ set))

Dalle specifiche (REQUIRES/EFFECTS):

pre_{super} = (size<capacity) $post_{super}$ = (retval==(elem ∈ set))

pre_{sub} = true $post_{sub}$ = (retval==(elem ∈ set))

Verifichiamo le regole:

- $pre_{super} \Rightarrow pre_{sub}$

  vero, poiché (size<capacity) ⇒ true è banalmente vera

- $(pre_{super}$ && $post_{sub}) \Rightarrow post_{super}$

  vero, poiché

  (size<capacity) && (retval==(elem ∈ set)) ⇒ (retval==(elem ∈ set))

  è anche questa banalmente vera

  (NOTA: il metodo della sottoclasse potrebbe fare qualcosa di diverso quando size>=capacity)

Regola delle proprietà

- la regola delle proprietà
  - il sotto-tipo deve preservare tutte le proprietà che possono essere provate sugli oggetti del super-tipo

- Ogni ragionamento sulle proprietà degli oggetti basato sul super-tipo è ancora valido quando gli oggetti appartengono al sotto-tipo
- Sono proprietà degli oggetti (non proprietà dei metodi)
- Da dove vengono le proprietà degli oggetti?
  - Dal modello del tipo di dato astratto (la struttura dati che l'oggetto rappresenta)
  - Esempio: proprietà dell'insieme di interi rappresentato da un oggetto IntSet

Esempi di proprietà di oggetti

Proprietà invarianti (esempi):

- un oggetto IntSet ha sempre elementi tutti diversi tra loro
- ad un oggetto FlexIntSet è sempre possibile aggiungere nuovi elementi (size < capacity)

Proprietà di evoluzione (esempi):

- il numero di elementi in un IntSet non può diminuire nel tempo
- se una chiamata add(n) restituisce true, da quel punto in poi contains(n) restituirà sempre true (stesso n)

Sono proprietà che possono essere individuate in fase di progettazione e incluse nella specifica della classe (in //OVERVIEW)

Verificare la regola delle proprietà

Per mostrare che un sotto-tipo soddisfa la regola delle proprietà dobbiamo mostrare che preserva le proprietà del super-tipo

Esempio: proprietà invariante di IntSet

- gli insiemi rappresentati hanno sempre elementi tutti diversi tra loro

Mostriamo che il sotto-tipo FlexIntSet verifica questa proprietà di IntSet...

- Insieme rappresentato come array
- Elementi diversi nell'array => elementi diversi nell'insieme rappresentato

```
public class FlexIntSet extends IntSet {

  // protected int[] a;  // EREDITATI
  // protected int size;

  public FlexIntSet(int capacity) { super(capacity); }

  public boolean add(int elem) {
    if (size>=a.length) {
      int[] tmp = new int[size+10];
      for (int i=0; i<size; i++)
        tmp[i] = a[i];
      a = tmp;
    }
    if (contains(elem)) return false;
    else { a[size]=elem; size++; return true; }
  }

  // public boolean contains(int elem)   // EREDITATO

}
```

Verificare la regola delle proprietà

Per most… …a la regola d… …rare che preserva…

Esempio: proprietà invariante di IntSet

- gli insiemi rappresentati hanno sempre elementi tutti diversi tra loro

Mostrian… …verifica questa p…

- Insier…
- Elementi diversi nell'array => elementi div…

Il costruttore crea un array vuoto, che rappresenta un insieme vuoto

- elementi tutti diversi: ok

Il metodo add aggiunge elementi solo se non presenti

- elementi tutti diversi: ok

Il metodo contains non modifica l'array

- elementi tutti diversi: ok

```
public class FlexIntSet extends IntSet {

  // protected int[] a;  // EREDITATI
  // protected int size;

  public FlexIntSet(int capacity) { super(capacity); }

  public boolean add(int elem) {
    if (size>=a.length) {
      int[] tmp = new int[size+10];
      for (int i=0; i<size; i++)
        tmp[i] = a[i];
      a = tmp;
    }
    if (contains(elem)) return false;
    else { a[size]=elem; size++; return true; }
  }

  // public boolean contains(int elem)    // EREDITATO
```

Regola delle proprietà verificata!

Invarianti e incapsulamento

La possibilità di dimostrare proprietà invarianti si basa fortemente sulla proprietà di incapsulamento della classe:

- La rappresentazione deve essere privata! Altrimenti tra una chiamata di metodo e l'altra un esterno potrebbe modificarla violando l'invariante
- Esempio: se l'array che rappresenta l'insieme fosse pubblico, dall'esterno si potrebbe aggiungere due copie dello stesso elemento (accedendo direttamente all'array)

Invarianti e incapsulamento

Quando lavoriamo variabili d'istanza di tipo non primitivo (oggetti) è comune commettere un errore di programmazione che porta ad esporre la rappresentazione, anche se privata

Esempio: aggiungiamo un paio di metodi a IntSet...

```
public class IntSet {

    private int[] a;
    private int size;

    public IntSet(int capacity) { ... }

    public int[] getElements() { return a; }   // nuovo metodo
    public int getSize() { return size; }      // nuovo metodo

    public boolean add(int elem) { ... }
    public boolean contains(int elem) { ... }

}
```

Che problema hanno questi nuovi metodi?

```
public class IntSet {

    private int[] a;
    private int size;

    public IntSet(int capacity) { ... }

    public int[] getElements() { return a; }   // nuovo metodo
    public int getSize() { return size; }      // nuovo metodo

    public boolean add(int elem) { ... }
    public boolean contains(int elem) { ... }

}
```

Che problema hanno questi nuovi metodi?

Il metodo getElements restituisce un riferimento all'array (che è un oggetto) consentendo al chiamante di modificarlo, anche se dovrebbe essere privato!!!

Errore grave che viola l'incapsulamento!!