Giorno 6
OOP: Principio di Sostituzione di Liskov

Sommario

1 Progettazione e Sviluppo di programmi Java

1.1 Fasi di Progettazione e Sviluppo

- Definizione della gerarchia di classi e interfacce
- Identificazione dei membri pubblici di ogni classe
- Definizione delle specifiche di ogni metodo pubblico (condizioni su parametri e risultato, comportamento atteso del metodo):
  - Aggiunta di commenti in cui scriviamo requirements ed effects
  - Esprimendo condizioni su parametri e variabili (e.g. **assert**)
- Testing delle singole classi
- Definizione dei membri privati seguendo il principio di incapsulamento
- Implementazione del codice, da verificare con i test già sviluppati

1.1.1 Esempio: IntSet

```
1  // OVERVIEW: un IntSet e' un insieme di interi
2     public class IntSet {
3         public IntSet(int capacity) {
4         // REQUIRES:    capacity non negativo
5         // EFFECTS:     crea un insieme vuoto che puo' ospitare capacity elementi
6     }
7     public boolean add(int elem) throws FullSetException {
8         // REQUIRES:    numero di elementi contenuti nell'insieme minore di capienza
9         // EFFECTS:     se elem non e' presente nell'insieme lo aggiunge e
      restituisce true,
10        //              restituisce false altrimenti
11    }
12    public boolean contains(int elem) {
13        // REQUIRES:
14        // EFFECTS:     restituisce true se elem p presente nell'insieme, false
      altrimenti
15    }
16 }
```

Nota: `add` può sollevare un'eccezione quando l'insieme è "pieno". Le eccezioni sono gestite in Java come in Javascript, e possono estendere Exception o RuntimeException. Per usare un'eccezione in una classe bisogna dichiararlo nell'intestazione.

JavaDoc Esiste una sintassi per le specifiche tramite commenti che permette di generare documentazione:

```
1  /**
2   * Aggiunge un elemento all'insieme
3   * @param elem valore intero
4   * @return true se l'inserimento viene aggiunto, false se gia' presente
5   * @throws FullSetException se l'insieme e' pieno
6   */
```

```
7 public boolean add(int elem) throws FullSetException {
8 ...
9 }
```

[Esempio di test e implementazione e altre cose, vd slide]
[vd. Infer, Pathfinder: model checker]

2 Principio di Sostituzione di Liskov

Principio di Sostituzione:

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

2.1 Differenza dalla subsumption

- La subsumption permette di considerare un tipo A tale che $A <: B$ come di tipo B.
- Il principio di sostituzione parla di **comportamento**, e.g. se si hanno due classi "`rettangolo`" e "`quadrato`", che hanno entrambe metodi per calcolare l'area, e `quadrato` <: `rettangolo`, allora: si può utilizzare un oggetto "`quadrato`" al posto di un `rettangolo`, ed il comportamento sarà indistinguibile da quello con un `rettangolo` con tutti i lati uguali.

Il principio di Liskov può o meno valere tra due classi, e verificare se vale è un **problema indecidibile**.

2.2 Regole indotte dal LSP

Il principio si traduce nella pratica in regole da seguire:

2.2.1 Regola della segnatura

- Gli oggetti del sottotipo devono avere tutti i metodi del supertipo
- Le segnature (signature) dei metodi del sottotipo devono essere compatibili con quelle corrispondenti del supertipo.

La presenza di tutti i metodi è **garantita dal compilatore Java** tramite i meccanismi di ereditarietà;
In caso di overriding, il metodo della sottoclasse deve:

- Avere la stessa firma del metodo della superclasse
- Sollevare meno eccezioni
- Avere un tipo di ritorno più specifico di quello della superclasse

2.2.2 Regola dei metodi

- Le chiamate dei metodi del sotto-tipo devono **comportarsi** come le chiamate dei corrispondenti metodi del supertipo

Devono valere le seguenti regole:

$$\text{precondizioni}_{super} \implies \text{precondizioni}_{sub}$$

$$(\text{precondizioni}_{super} \wedge \text{postcondizioni}_{sub}) \implies \text{postcondizioni}_{super}$$

Dove le **postcondizioni** altro non sono che gli **effetti** del metodo.

Ciò significa che **le precondizioni possono essere indebolite nel sottotipo, e le postcondizioni possono essere rafforzate.**

Esempio: Si hanno due classi: La **superclasse** `IntSet` e la sua **sottoclasse** `FlexIntSet` (rispettivamente insieme di interi di cardinalità fissata e insieme flessibile di interi):

Superclasse IntSet

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

Ossia la precondizione è $size < capacity$, mentre la postcondizione è $retval \in set$

Sottoclasse FixedIntSet

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

Le precondizioni sono vuote, quindi $true$, mentre le postcondizioni sono sempre $retval \in set$.

Di conseguenza:

$$(size < capacity) \implies true \qquad (size < capacity) \wedge (retval \in set) \implies (retval \in set)$$

Sono entrambe vere.

Nota: perché seconda parte della regola? Ci chiediamo a cosa serve $\text{precondizioni}_{super}$ nella regola:

$$(\text{precondizioni}_{super} \wedge \text{postcondizioni}_{sub}) \implies \text{postcondizioni}_{super}$$

Nell'esempio sopra, **dobbiamo poter gestire il caso** $capacity > size$; in tal caso il metodo del supertipo lancerà qualche eccezione forse, mentre il sottotipo non ha problemi a continuare ad aggiungere: non ha precondizioni!

2.2.3 Regola delle proprietà

- Il sotto-tipo deve preservare tutte le proprietà che possono essere provate sugli oggetti del supertipo.

Ciò significa che i **ragionamenti** sulle proprietà degli oggetti del supertipo sono ancora validi su quelle del sottotipo;

Esempi di proprietà Sempre rispetto all'esempio del punto precedente

Proprietà invarianti:

- IntSet ha sempre elementi diversi
- Ad un oggetto FlexIntSet è sempre possibile aggiungere nuovi elementi

Proprietà di evoluzione:

- Il numero di elementi in IntSet non può diminuire nel tempo
- Se $add(n) \to true$, allora da quel punto in poi $contains(n) \to true$.

Invarianti e incapsulamento

- La rappresentazione deve essere privata! Altrimenti dall'esterno si potrebbe modificare qualcosa modificando l'invariante!

ERRORE COMUNE Anche se la rappresentazione di un attributo è privata, questo potrebbe essere un oggetto (e.g. un array), e se per caso questo oggetto fosse ritornato da una funzione, sarebbe **passato per riferimento**, errore grave che viola l'incapsulamento.

```
public class IntSet {
    private int[] a;
    public int[] getElements() { return a; } // ERRORE GRAVE!!!

...
}
```