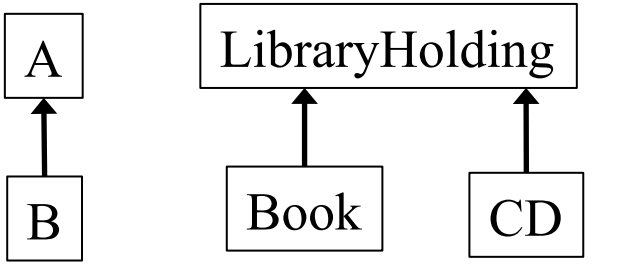
Il principio di sostituzione

Cosa intendiamo per “sottotipo”

Volgata “*ogni entità B è anche A”*

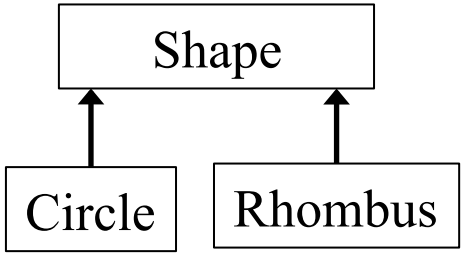
- Esempio: Contenuti Libreria LH
  - Ogni Book è un LH
  - Ogni CD è un LH



La nozione di sottotipo

- “*B è un sottotipo di A*” :

  “ogni oggetto che soddisfa le regole di comportamento di B
  soddisfa anche le regole di A”



Visione da programmatore: Il codice scritto usando la specifica di A si comporta correttamente anche quando viene usato come B

- Tanti vantaggi: disegno pulito, test condivisi, riuso del codice …

Sottotipi e principio di sostituzione

I sottotipi possono essere sostituiti ai supertipi

- Una istanca del sottotipo non genera un fallimento dato che rispetta tutte le proprietà attese del supertito
- Cliente non vede la differenza

Una possibile definizione:

B viene detto ***true subtype*** di A se B ha una specifica più vincolatante rispetto alla specifica di A

- Questa proporietà ***non*** corrisponde alla nozione di ***Java* subtype**
- Java subtype non rispetta la nozione di true subtype

Subtyping vs. subclassing

Sottotipi (Subtype) — una nozione semantica

- B è un sottotipo di A se e solo un oggetto di tipo B può stare al posto di (mascherarsi come) A in un qualunque contesto
- I comportamenti osservabili di B sono un sottoinsiuene di quelli di A

Ereditarietà (subclass) — è una nozione di implementazione

- Permette la fattorizzazione e il riuso del codice
- Una nuova classe è creata per differenza dalla classe padre

Java mette assieme le due nozioni:

- In Java ogni sottoclasse diventa anche un sottotipo
- Ma non è un true subtype

Aggiungere funzionalità via ereditarietà

Back-end di un sito web di vendita … la classe *products…*

```
class Product {
    private String title;
    private String description;
    private int price; // in una qualche valuta
    public int getPrice() {
        return price;
     }
    public int getTax() {
        return (int)(getPrice() * 0.096);
    }
    …
}
```

... Adesso entriamo nella fase dei saldi

Cut&Paste: evitare

Mai scrivere una cosa del genere:

```
class SaleProduct {
    private String title;
    private String description;
    private int price;
    private float factor;
    public int getPrice() {
       return (int)(price*factor);
    }
    public int getTax() {
        return (int)(getPrice() * 0.096);
    }
    …
}
```

Ereditarietà .. Una scelta migliore


```
class SaleProduct extends Product {
    private float factor;
    public int getPrice() {
      return (int)(super.getPrice()*factor);
    }
}
```

Benefici (di nuovo…)

- Si ereditano automaticamente tutte le caratteristiche non modificate (variabili di istanza, metodi, …)
  - Nell'implementazione
  - Nella specifica
  - Modularità: solo le differenze

- Riuso del codice
  - Il codice del cliente non cambia a causa delle funzionalità aggiunte

Un esempio significativo: square vs rectangle


```
interface Rectangle {
  // @effects: definisce le dimensione della figura:
  //           this_post.width = w, this_post.height = h
  void setSize(int w, int h);
}
interface Square extends Rectangle {…}
```

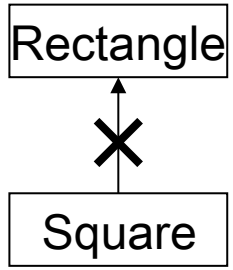
Specifichiamo **`setSize`** di **`Square`**

```
1. // @requires: w = h
   // @effects: definisce la dimensione
  void setSize(int w, int h);
2.// @effects: assegna il valore del lato del quadrato
  void setSize(int edgeLength);
3.// @effects:  assegna w a this.width && this.height
  void setSize(int w, int h);
4. // @effects: definisce le dimensioni
   // @throws BadSizeException if w != h
  void setSize(int w, int h) throws BadSizeException;
```

Square vs Rectangle

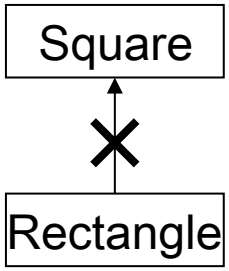
`Square` non è un true subtype di **`Rectangle`**:

- **`Rectangle`**: gli oggetti istanza della classe hanno una base e una altezza (width && height) che possono essere mofificate singolarmente
- **`Square`**: viola questa proprietà



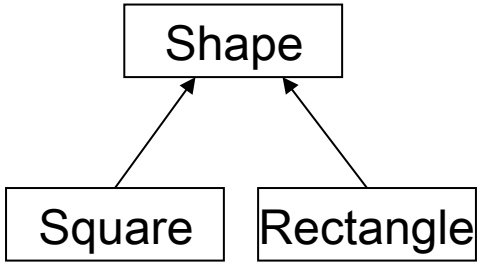
`Rectangle Square` non è un true subtype di **`Square`**:

- **`Square`**: base e altezza sono identiche
- **`Rectangle:`** viola questa proprietà



Soluzione: sono nozioni differenti

- Non sono collegate



JDK: aspetti critici


```
class Hashtable<K,V> {
  public void put(K key, V value){…}
  public V get(K key){…}
}

// Keys && values: hanno tipo string
class Properties extends Hashtable<Object,Object> {
   public void setProperty(String key, String val) {
     put(key,val);
   }
   public String getProperty(String key) {
     return (String)get(key);
   }
}
```

```
Properties p = new Properties();
Hashtable tbl = p;
tbl.put("One", 1);
p.getProperty("One"); // oops!
```

Perchè? Violazione RepInv

`Properties` RepInv:

- Keys && values sono oggetti di tipo **`String`**

Cliente puo usare **`Properties`** come una normale **`Hashtable`**

- Inserita in un contesto può determinare la violazione del *rep invariant*

DaJavadoc:

> *Because Properties inherits from Hashtable, the put and putAll methods can be applied to a Properties object. ... If the store or save method is called on a "compromised" Properties object that contains a non-String key or value, the call will fail.*

#1: Usare meglio Java Generics

Scelta errata:

```
class Properties extends Hashtable<Object,Object>
   { …
}
```

Scelta migliore:

```
class Properties extends Hashtable<String,String>
   { …
}
```

JDK: non lo fa automaticamente. Perche?

- Backward-compatibility (al solito ….)

#2: Composizione


```
class Properties {
   private Hashtable<Object, Object> hashtable;

   public void setProperty(String key, String value) {
      hashtable.put(key,value);
   }

   public String getProperty(String key) {
      return (String) hashtable.get(key);
   }

   …
}
```

Principio di sostituzione: Proprietà

Se B è un sottotipo di A, allora B può essere sempre sostiture A (ogni istanza di B può stare in tutti i contesti dove può stare una istanza di A)

Tutte le proprietà garantiite da A devono essere garantite da B

B può rafforzare le proprietà o introdurne delle nuove

- Si possono introdurre nuovi metodi (che comunque preservano RepInv))

B non può indebolire una specifica

- Non e' possibile rimuovere metodi

Principio sostituzione: metodi

Metodi del supertipo sono ereditati senza modifiche o riscritti.

Ogni metodo riscritto deve rafforzare la specifica:

- Non deve fare richieste extra al cliente (“weaker precondition”)
  - *La clausola @requires deve essere al **massimo** stringente quanto quello del metodo del supertipo*
- Deve garantire almeno le medesime proprietà (“stronger postcondition”)
  - *La clausola @effett deve essere stringente **almeno** quanto quella del supertipo*
  - Non si può aggiungere entità nella clausola @modifies
  - La clausola @return deve promettere almeno le stesse cose del supertipo
  - *La clausola @throws* deve indicare gli stessi (o meno) tipi di eccezioni.

Principio sostituzione: segnatura

Metodi: parametri input:

- Il tipo degli argomenti dei metodi in A può essere sostituito da supertipi in B ("contravariance")
- Poi vediamo cosa fa Java

Metodi result:

- Il tipo del risultato dei metodi in A può essere un sottoyipo in B ("covariance")
- Non possono essere introdotte nuove eccezioni
- Le eccezioni esistenti possono essere sotituite da sotto-tipi

Principio di sostituzione: riepilogo

- Devono essere supportate
  - la regola della segnatura
    - ✓ gli oggetti del sotto-tipo devono avere tutti i metodi del super-tipo
    - ✓ le segnature 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
- NB: le regole riguardano la semantica!

Regola della segnatura (Java)

- Se una chiamata è type-correct per il super-tipo lo è anche per il sotto-tipo
  - garantita dal compilatore Java
    - che permette che i metodi del sotto-tipo sollevino meno eccezioni di quelli del super-tipo
    - da Java 5 un metodo della sotto-classe può sovrascrivere un metodo della super-classe con la stessa firma fornendo un return type più specifico
      - docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.4.8.3
- le altre due regole non possono esser garantite dal compilatore Java…
  - dato che hanno a che fare con la specifica della semantica!

Regola dei metodi (Java)

- Si può ragionare sulle chiamate dei metodi usando la specifica del super-tipo anche se viene eseguito il codice del sotto-tipo
- Si è garantiti che va bene se i metodi del sotto-tipo hanno esattamente le stesse specifiche di quelli del super-tipo
- Come possono essere diverse?
  - se la specifica nel super-tipo è non-deterministica (comportamento sotto-specificato) il sotto-tipo può avere una specifica più forte che risolve (in parte) il non-determinismo

Regola dei metodi

- In generale un sotto-tipo può indebolire le pre-condizioni e rafforzare le post-condizioni
- Per avere compatibilità tra specifiche del super-tipo e del sotto-tipo devono essere soddisfatte le regole
  - regola delle pre-condizione
    - ✓ pre_{super} ==> pre_{sub}
  - regola delle post-condizione
    - ✓ pre_{super} && $post_{sub}$ ==> $post_{super}$

Regola dei metodi

- Ha senso indebolire la precondizione
  - pre $_{super}$ ==> pre $_{sub}$

  perché il codice che utilizza il metodo è scritto per usare il super-tipo
  - ne verifica la pre-condizione
  - verifica anche la pre-condizione del metodo del sotto-tipo
- Esempio: un metodo in IntSet

```
public void addZero(  )
    // REQUIRES: this non e' vuoto
    // EFFECTS: aggiunge 0 a this
```

potrebbe essere ridefinito in un sotto-tipo

```
public void addZero(  )
    // EFFECTS: aggiunge 0 a this
```

Regola dei metodi

- Ha senso rafforzare la post-condizione
  - pre $_{super}$ && post $_{sub}$ ==> post $_{super}$

  perché il codice che utilizza il metodo è scritto per usare il super-tipo
  - assume come effetti quelli specificati nel super-tipo
  - gli effetti del metodo del sotto-tipo includono comunque quelli del super-tipo (se la chiamata soddisfa la pre-condizione più forte)
- Esempio: un metodo in IntSet

```
public void addZero(  )
     // REQUIRES: this non e' vuoto
     // EFFECTS: aggiunge 0 a this
```

potrebbe essere ridefinito in un sotto-tipo

```
public void addZero(  )
     // EFFECTS: se this non e' vuoto aggiunge 0 a this
     // altrimenti aggiunge 1 a this
```

Regola dei metodi: violazioni

- Consideriamo `insert` in `IntSet`

```
public class IntSet {
    public void insert(int x)
           // EFFECTS: aggiunge x a this
```

- Supponiamo di definire un sotto-tipo di `IntSet` con la seguente specifica di `insert`

```
public class StupidIntSet extends IntSet {
    public void insert(int x)
           // EFFECTS: aggiunge x a this se x è pari,
           // altrimenti non fa nulla
```

Regola delle proprietà

- Il 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
    - ✓ le proprietà degli insiemi matematici, etc.
    - ✓ le elenchiamo esplicitamente nell'overview del super-tipo
      - un tipo astratto può avere un numero infinito di proprietà
- Proprietà invarianti
  - un FatSet non è mai vuoto
- Proprietà di evoluzione
  - il numero di caratteri di una String non cambia

Regola delle proprietà

- Per mostrare che un sotto-tipo soddisfa la regola delle proprietà dobbiamo mostrare che preserva le proprietà del super-tipo
- Per le proprietà invarianti
  - bisogna provare che creatori e produttori del sotto-tipo stabiliscono l'invariante (solita induzione sul tipo)
  - che tutti i metodi (anche quelli nuovi, inclusi i costruttori) del sotto-tipo preservano l'invariante
- Per le proprietà di evoluzione
  - bisogna mostrare che ogni metodo del sotto-tipo le preserva

Regola delle proprietà: una proprietà invariante

- Il tipo `FatSet` è caratterizzato dalla proprietà che i suoi oggetti non sono mai vuoti

```
`// OVERVIEW: un FatSet e' un insieme modificabile di interi
 // la cui dimensione e' sempre almeno 1
```

- Assumiamo che `FatSet` non abbia un metodo remove, ma invece abbia un metodo `removeNonEmpty`

```
public void removeNonEmpty (int x)
     // EFFECTS: se x e' in this e this contiene altri elementi
     // rimuovi x da this
```

e abbia un costruttore che crea un insieme con almeno un elemento. Si può provare che gli oggetti `FatSet` hanno dimensione maggiore di 0?

Regola delle proprietà: una proprietà invariante

- Consideriamo il sotto-tipo ThinSet che ha tutti i metodi di FatSet con identiche specifiche e in aggiunta il metodo

```
public void remove(int x)
    // EFFECTS: rimuove x da this
```

- ThinSet non è un sotto-tipo legale di FatSet
  - perché il suo extra metodo può svuotare l'oggetto, e
  - l'invariante del super-tipo non sarebbe conservato

Una proprietà di evoluzione (non modificabilità)

- Il tipo SimpleSet ha i due soli metodi insert e isIn
  - gli oggetti di SimpleSet possono solo crescere in dimensione
  - IntSet non può essere un sotto-tipo di SimpleSet perché il metodo remove non conserva la proprietà