mapio – Orali del 2020-06-17

January 20, 2021

Se si implementa `equals`, bisogna implementare anche `hashCode` o i metodi delle collezioni potrebbero comportarsi in modo strano. Nella classe `Rettangolo`, le variabili di istanza ne rappresentano le coordinate in basso a sinistra ed in alto a destra: anche questo aspetto è parte dell'invariante di rappresentazione, e va quindi controllato nel costruttore (ed eventualmente in `repOk`). Il codice che restituisce il *bounding box* deve anche controllare se l'insieme di coordinate ricevuto in input è vuoto, in particolare se si effettua un *priming* delle variabili che vanno poi a comporre il *bounding box*. Altrimenti si otterrebbe un *bounding box* degenere.

[ ]:
```java
public static Rettangolo boundingBox(Set<Coordinata> coordinate) {
    Objects.requireNotNull(coordinate,
                           "L'insieme delle coordinate non può essere nullo");

    int maxX = Integer.MIN_VALUE;
    int maxY = Integer.MIN_VALUE;
    int minX = Integer.MAX_VALUE;
    int minX = Integer.MAX_VALUE;

    for (Coordinata c : coordinate) {
        if (c.x > maxX)
            maxX = c.x;
        if (c.y > maxY)
            maxY = c.y;
        if (c.x < minX)
            minX = c.x;
        if (c.y < minY)
            minY = c.y;
    }

    return new Rettangolo(new Coordinata(minX, minY),
                          new Coordinata(maxX, maxY));
}
```

Si potrebbe quindi precisare la specifica, dicendo che il *bounding box* di un insieme vuoto di coordinate non esiste e sollevando un'eccezione opportuna.

[ ]:
```java
private Set<Coordinata> coordinate;

public Set<Coordinata> coordinate() {
```

```
    return new HashSet(coordinate);
}
```

Restituire un nuovo insieme di coordinate ha senso, perché così facendo evitiamo che chi invoca `coordinate` non entri in possesso di un riferimento alla rappresentazione della classe (data dal membro privato). Anche se questa fosse `final` ha senso restituire un nuovo insieme, perché `final` non impedirebbe a chi invoca `coordinate` di svuotare l'insieme (per esempio). Il costruttore usato sopra si dice "costruttore copia," ma c'è un'altra possibilità meno onerosa. `Set` è una classe o un'interfaccia? Un'interfaccia. Sarebbe difficile implementarla in modo tale da rendere i suoi oggetti immutabili? Si potrebbero modificare i metodi mutazionali così da sollevare un'eccezione. La possibilità meno onerosa è quella di utilizzare `ImmutableSet`, un metodo statico di fabbricazione che restituisce lo stesso insieme ricevuto in input ma immutabile.

Cos'è il principio di sostituzione di Liskov? Come lo si garantisce? Perché deve valere? In un linguaggio orientato agli oggetti, deve esserci una relazione di supertipo e sottotipo: il principio di sostituzione si riferisce quindi alla possibilità di sostituire un sottotipo al posto del suo supertipo senza che avvengano problemi. Questi vengono evitati solo durante la compilazione? No, anche durante l'esecuzione. Ci sono tre cose da controllare affinché questo principio valga: la regola della segnatura, la regola dei metodi, la regola della proprietà. Cosa dicono? - Segnatura: un sottotipo deve presentare tutti i metodi del supertipo con la stessa segnatura, altrimenti sostituendo un sottotipo al supertipo il programma non verrebbe compilato. In Java bisogna stare attenti a questa caratteristica, oppure c'è un meccanismo che ne garantisce la validità? Sì, l'ereditarietà: un sottotipo eredita tutti i metodi del supertipo, quindi questa regola è sempre vera. - Metodi: la semantica dei metodi non può cambiare drasticamente tra sottotipo e supertipo. Poiché richiede una comprensione del codice, non può essere garantita dal compilatore. - Proprietà: le pre-condizioni di un metodo sovrascritto dal sottotipo possono essere meno restrittive di quelle del supertipo, mentre le post-condizioni possono essere più stringenti. Il metodo `add` di `Set` funziona correttamente, aggiungendo un elemento all'insieme; il metodo `add` di `ImmutableHashSet` solleva un'eccezione: questo viola il principio di sostituzione? Sì, perché lo stesso metodo ha due comportamenti diversi nel sottotipo e nel supertipo. È per questo che la documentazione di `Set` indica i metodi mutazionali come 'opzionali.'

Cosa si intende per "polimorfismo?" Quali esempi ci sono? In Java, questo aspetto si manifesta nei generici: le operazioni di un insieme di stringhe sono le stesse di un insieme di interi, così mi evito di dover riscrivere lo stesso codice. Si dice anche che Java supporta il polimorfismo dei metodi, oppure che le variabili in Java sono polimorfe: cosa si intende con queste espressioni? Una variabile dal tipo apparente `Set` è polimorfa, perché il suo tipo concreto può essere `HashSet`, `TreeSet`, eccetera. Quindi la variabile di un certo supertipo può contenere diversi sottotipi. Per quanto riguarda i metodi, il polimorfismo si manifesta nella possibilità di 'sovraccaricarli:' cosa significa? In quali casi può essere utile, tenendo fisso il numero di argomenti del metodo sovraccaricato? Un metodo astratto che calcola l'area di una figura geometrica, e le sue implementazioni concrete appartenenti a classi come *Quadrato*, *Rettangolo*, *Cerchio*, non è un buon esempio perché qui il metodo astratto viene **sovrascritto**, non **sovraccaricato**.

Cosa si intende per "insieme specificando," o "*specificand set*?" Com'è costruita una specifica rispetto al suo *specificand set*? Se nella specifica mi concentrassi troppo su come implementare un metodo, questa risulterebbe essere troppo restrittiva e potrebbe tagliar via delle implementazioni accettabili. Quale può essere un esempio concreto? Cosa sarebbe cambiato se nella specifica di `Rettangolo` si fosse parlato di "lista di coordinate" anziché di "insieme di coordinate?" Che

caratteristiche ha una lista rispetto ad un insieme che può essere fastidioso implementare? Dovrei gestire i duplicati, ma anche la sua indicizzazione (il fatto che ci sia un primo, un secondo, eccetera).

[ ]:
```java
public class TeraminoClass implements Teramino {
    public final char nome;
    // Non c'è motivo di non usare `final` anche per le seguenti
    private Set<Coordinata> coordinate;
    private char tipo;
    private Rettangolo boundingBox;

    public TeraminoClass(Set<Coordinata> s, char nome) {
        if (s == null)
            throw new NullPointerException("Fornire delle coordinate adeguate");

        this.nome = nome;
        coordinate = s;
        boundingBox = Rettangolo.creaRettangolo(s);
    }
}
```

Anche nella precedente porzione di codice, sarebbe stato il caso di controllare che `s` non sia vuoto. Inoltre, l'istruzione `coordinate = s;` copia il riferimento a `Set<Coordinata>` in `coordinate`: questo è un problema, perché `boundingBox` dovrebbe riferirsi sempre all'insieme di coordinate del teramino. Poiché il teramino diviene mutabile (`coordinate` può essere modificato in qualunque momento, visto che l'utente della classe ha in mano il suo riferimento), bisognerebbe calcolare `boundingBox` in ogni metodo che modifica il teramino. In generale, tenere informazioni duplicate circa lo stato in più di un metodo è indice di un potenziale problema: non devo solo garantire che `coordinate` non si svuoti e che sia diverso da `null`, ma anche che le altre variabili associate ad essa (in questo caso `boundingBox`) siano mantenute sempre e comunque sincronizzate con `coordinate`.

[ ]:
```java
public Set<Coordinata> coordinate() {
    return coordinate;
}
```

In questo modo si espone la rappresentazione: l'utente di questa classe ha quindi due diverse occasioni per romperla.

[ ]:
```java
public Rettangolo boundingBox() {
    return boundingBox;
}
```

Questo invece non è un problema: rivelare parte della rappresentazione va bene a patto che questa non possa essere alterata.

[ ]:
```java
public static Teramino teramino(char nome, char tipo, int rotazioni) {
    Set<Coordinata> c = new HashSet<>();
    nome = nome;
    tipo = tipo;
```

```
    Teramino t = new TeraminoClass(c, 'A');
    Teramino result = new TeraminoClass(c, 'A');

    // …

    for (int i = 0; i < rotazioni; i++)
        result = t.ruota();

    return result;
}
```

Le istruzioni `nome = nome` e `tipo = tipo` non hanno alcun effetto: come si possono correggere? Non posso scrivere `this.nome = nome` perché in un metodo statico non ho riferimenti all'istanza dell'oggetto. Potrei scrivere, dopo aver instanziato `t`, `t.nome = nome` o ci sarebbe qualche problema? Ovviamente non dovrei potere, dato che la specifica vuole un nome immutabile per la classe `Teramino`. Il ciclo dopo l'ellissi funziona o ha un problema? Non funziona perché, quale che sia il valore di `rotazioni` (maggiore di zero), `result` è il teramino `t` ruotato una sola volta. Si può correggere sostituendo l'istruzione interna con `t = t.ruota();`, visto che `Teramino` è una classe immutabile.

Cos'è la funzione di astrazione? È una funzione, quindi qual è il suo dominio? Qual è il suo codominio? Ce n'è una per ogni oggetto. Il suo dominio è il prodotto cartesiano di tutti i possibili valori dei suoi membri, e non ha niente a che fare con il sorgente: questo va infatti 'fissato' per poter parlare della funzione di astrazione. Perché in generale è una funzione molti-a-uno? Com'è nel caso della classe `Coordinata`? È biiettiva, perché a valori dei membri diversi corrispondono coordinate diverse. Qual è un esempio di oggetto la cui funzione di astrazione è molti-a-uno? Perché quella di `TeraminoClass` è molti-a-uno? Gli esempi sono fondamentali per dimostrare di aver capito.

A cosa servono le interfacce di Java? Cosa ci può stare sotto un'interfaccia? Ci può stare di tutto, anche le interfacce. Qual è il senso che avere un interfaccia con un'altra interfaccia come sottotipo? Può aver senso se rende più specifico il contratto espresso dal suo supertipo. Perché si chiamano 'sottotipi' quando tipicamente sono più specializzati del supertipo? Un sottinsieme ha meno elementi dell'insieme di partenza, mentre qui un sottotipo **sembrerebbe** avere più metodi (e quindi più elementi, se vedessimo il supertipo come l'insieme e il sottotipo come un sottinsieme). Aumentando l'insieme di metodi, la quantità di implementazioni che soddisfano un sottotipo diventa minore di quelle che soddisfano il supertipo. Allo stesso modo, se definisco un sottinsieme in base alle proprietà che i suoi elementi devono soddisfare, più proprietà esprimo più il sottinsieme è piccolo. Il fatto che le interfacce possano contenere metodi di default a cosa le fa assomigliare? Che differenza c'è tra interfacce e classi astratte? Che nelle classi astratte si possono avere attributi, nelle interfacce no. È per questo che i metodi di default di un'interfaccia sono l'unica via per modificare lo stato dell'oggetto. Quindi non posso fare affidamento sul fatto che chi implementa l'interfaccia abbia un certo attributo. Viceversa, le classi astratte possono avere attributi sui quali i suoi metodi o quelli delle sottoclassi hanno modo di agire (a meno di restrizioni sulla visibilità). Perché questo rende preferibili in qualche caso le interfacce, ed in qualche caso le classi astratte? Ha senso usare una classe astratta quando suppongo che le classi concrete che la estendono condividono un certo attributo: in questo caso, come va dichiarato tale attributo nella classe astratta? `protected`. Estendere una classe astratta con attributi `protected` può esporre a dei rischi? Un attributo `private` in una classe astratta può essere visto solo da questa, e l'unico modo per renderlo accessibile alle sottoclassi è implementare un *getter*. Il punto cruciale è che inserire attributi nella classe

astratta 'inchioda' una specifica implementazione: per esempio, se abbiamo deciso di utilizzare una lista come attributo comune nella classe astratta, chiunque decide di estenderla si trova a dover lavorare con una decisione implementativa non sua. Inoltre, può capitare che chi la estende interagisca in modo pericoloso con il membro a cui ha accesso. Nell'interfaccia sono limitato ai metodi, e ne posso contrattualizzare il comportamento nelle specifiche; nella classe astratta dovrei invece documentare il codice in direzione di un'eventuale estensione, fermo restando che nulla vieta all'estensore della classe astratta di 'rompere' i metodi la superclasse: è per questo che l'invariante di rappresentazione delle sottoclassi deve sempre tener conto di quello della superclasse. Aumentare le funzionalità di un oggetto estendendo una superclasse anziché implementando un'interfaccia ha quindi il vantaggio di condividere codice e dati, ma lo svantaggio che un membro `protected` può essere 'spaccato' da chiunque estenda la superclasse (e questo si ripercuote non solo nella sottoclasse che estende, ma anche nella superclasse il che si traduce in ovvi problemi di debugging). Viceversa, implementare un'interfaccia richiede la stesura di più codice ma garantisce una maggior protezione contro eventuali fraintendimenti della documentazione.

Cosa sono le procedure totali e parziali? Quando le incontriamo? La totalità è una proprietà necessaria? Una procedura è totale se il suo comportamento è specificato per tutti gli input. Se una procedura è parziale, devo sollevare un'eccezione per la parte di input che questa non si aspetta? Non c'è verso di specificare totalmente alcune procedure, per esempio nel caso di radice quadrata da reali positivi a reali. Viceversa, la somma di due numeri è sempre totale. Quale delle due alternative è meglio? Quella totale. È conveniente obbligare un programmatore a scrivere solo funzioni totali? È sempre possibile capire **al volo** se l'input di un procedura è nel dominio per cui il comportamento è specificato? Che esempio si può fare? Una ricerca binaria che restituisce l'indice dell'elemento desiderato o $ -1 $ se non lo trova è totale o parziale? È parziale, perché la specifica non indica cosa accade in caso di vettori non ordinati. Come posso renderla totale? È possibile verificare che il vettore è ordinato? È un bene averla resa totale? Un punto cruciale del software è impedire il proseguimento dell'esecuzione in caso di errore. Il costo per rendere una procedura totale può essere più 'grande' di quello per farla funzionare, quindi potrebbe non aver senso. Nel caso della ricerca binaria, un'implementazione totale richiede $ O (n) $ per verificare se il vettore è ordinato e di fatto si annullerebbero i vantaggi associati ad un eventuale vettore ordinato. Tale procedura può restare parziale, a patto di essere accompagnata da una documentazione robusta e che l'utente possa verificare se il vettore è ordinato prima della ricerca binaria.

[ ]:
```java
public MyTeramino(char n, List<Coordinata> coords) {
    nome = n;
    Objects.requireNonNull(coords);

    for (Coordinata c : coords) {
        Objects.requireNonNull(c);
        // …
    }
}
```

Nel ciclo ha senso controllare che ciascuna `c` sia diversa da `null`, perché in una lista di oggetti è possibile inserire qualsiasi cosa. Un possibile esempio di funzione di astrazione non iniettiva è mappare il teramino contenente le coordinate `(0, 0)` e `(0, 0), (0, 0)` nello stesso oggetto astratto, come può accadere usando una lista anziché un insieme per rappresentare questa classe.

[ ]:
```java
public Rettangolo boundingBox() {
    Rettangolo x = Rettangolo.boundingBox(componenti);

    return x;
}
```

Calcolo 'pigro:' anziché calcolare il *bounding box* ogni volta che questo metodo viene invocato, si può tenere da parte un attributo mutabile `Rettangolo` nella classe `MyTeramino` a cui si assegna inizialmente `null`. All'invocazione di `boundingBox`, se tale attributo è diverso da `null` se ne restituisce il valore; altrimenti, si calcola il *bounding box*, lo si assegna all'attributo e poi se ne restituisce il valore. In questo modo non si spreca tempo se nessuno richiede il *bounding box*, come può invece succedere andando a calcolarlo nel costruttore, mentre se viene richiesto allora la classe "se lo ricorda" inserendolo nell'attributo. L'unico svantaggio di questa soluzione si manifesterebbe se un teramino fosse mutabile, in quanto si dovrebbe invalidare l'attributo associato al *bounding box* a seguito di ogni metodo mutazionale. Quindi 'cachare' un'informazione è una scelta da fare con attenzione.

Cosa sono le eccezioni? Quando si adoperano? Che tipo di eccezioni esistono? Come può avvenire una *graceful degradation* associata al sollevamento di un'eccezione? Ad esempio, nel caso di un browser, anziché terminare l'esecuzione dopo non essere riuscito a trovare una pagina, dà la possibilità di ricaricarla. Se invece si sta scrivendo su un file e lo spazio su disco termina, la causa esterna dell'eccezione è "più irreversibile" della precedente ed una *graceful degradation* è più difficile da implementare. Il compilatore riconosce ed impone la gestione delle eccezioni *checked*, ma non di quelle *unchecked*. Anche una *unchecked* può terminare con una *graceful degradation*? Sì. Per esempio, se non si è molto sicuri di una porzione di codice, è possibile avvolgerla in un *try-catch*. Se questo codice appartenesse a Microsoft Word, il programma potrebbe avvisare l'utente di un errore imprevisto ed offrirgli di salvare il file. Un buon esempio per parlare di *graceful degradation* è la rete che viene a mancare e torna in un secondo momento.

Cos'è l'induzione sui tipi di dato?

Perché in Java si tende ad usare molto l'iterazione esterna e poco quella interna? La Liskov parla di appropriatezza delle specifiche: può essere appropriato che un insieme abbia un metodo che restituisca la somma dei suoi elementi? L'iterazione esterna è utile per arricchire le funzionalità di certe collezioni di oggetti senza appesantirne la specifica con metodi di iterazione interna. Qual è lo scopo della classe interna a cui la Liskov si riferisce con il termine di '*generator*?' Perché fare una classe innestata? La classe interna ha visibilità dei membri della classe esterna, la cui rappresentazione non viene esposta. Se tali membri sono mutabili, la classe esterna si deve preoccupare di proteggerli. La classe interna ha comunque padronanza completa dell'implementazione usata da quella esterna.