mapio – Orali del 22.01.2021

February 3, 2021

Il seguente `repOk` della classe `Directory` restituisce sempre `true`?

[ ]:
```java
@Override public boolean repOk() {
  int sommaDimensione = 0;

  for (Entry e : contenuto)
    sommaDimensione += entry.dimensione();

  for (int i = 0; i < contenuto.size(); i++)
    for (int j = 0; j < contenuto.size(); j++)
      if (i != j && contenuto.get(i).equals(contenuto.get(j)))
        return false;

  return true;
}
```

Solito problema: effettuando il *caching* della dimensione di `this`, se inserisco un file in una sottocartella di `this`, quest'ultima non ha modo di aggiornare la sua dimensione correttamente. L'implementazione di `repOk` è corretta, ed il metodo viene invocato nei posti giusti. Tuttavia esiste una categoria di oggetti per cui questo approccio non è sufficiente: quali? Quelli per cui l'invariante di rappresentazione non dipende solo dai loro attributi: nel nostro caso, l'invariante di `this` dipende anche dalla dimensione delle sue sottocartelle. Fare *caching* di un'informazione correttamente è molto complicato. Perché il seguente metodo di `Filesystem` dà errore di compilazione?

[ ]:
```java
public Entry ottieni(Path p) {
  return new Entry("entry");
}
```

Perché `Entry` è una classe astratta, quindi non può essere istanziata. Cos'è l'astrazione dei dati secondo la Liskov? Qual è l'obbiettivo? Quali meccanismi abbiamo per realizzare questa astrazione? Non è un oggetto ad essere adeguato, semmai è la specificazione. Quali benefici otteniamo tramite l'astrazione per specificazione? Cosa si intende per località e modificabilità? Che differenza c'è tra procedura e metodo? È vero che per verificare la correttezza di un metodo è sufficiente riflettere sulla sua implementazione? No, perché questa dipende dallo stato dell'oggetto: sarebbero quindi necessarie tecniche di dimostrazione più complicate. In che senso «il codice è all'interno di una procedura e quello fuori non la riguarda»? (Domanda retorica) Cosa sono i tipi apparenti ed i tipi concreti? Perché sono concetti importanti? —

Perché l'attributo `nome` di `Entry` è `protected`, se poi metto a disposizione un *getter* pubblico?

Quando progettiamo per l'estensione, bisogna prestare attenzione ai modificatori di visibilità: può aver senso avere un attributo protetto se lo utilizziamo nelle sottoclassi, ma non se mettiamo a disposizione un metodo pubblico per accedervi. Per quale motivo può essere utile rendere un attributo privato anche se è finale? (Analogamente, per quale ragione può essere svantaggioso rendere l'attributo visibile a chi estende la classe?) In generale si preferisce precludere alle sottoclassi la possibilità di modificare lo stato della classe, e nascondere dettagli implementativi: così facendo, la classe in questione gode della proprietà di località e modificabilità (posso cambiarne la rappresentazione senza intaccare le sottoclassi). In questo caso particolare ha senso tenere `nome` pubblico, perché non ci sono tanti altri modi per rappresentarlo se non una stringa; questo approccio può essere inoltre ragionevole se gli aspetti implementativi che rivelo offrono grandissimi vantaggi alle sottoclassi. L'invariante di rappresentazione della classe `Directory` è corretto? È necessario, o è anche sufficiente?

[ ]:
```
private int dimensione;
private List<Entry> entries;

/*
** REP INV: dimensione >= 0
**    […]
*/
```

Sicuramente è necessario che la dimensione non sia un numero negativo. Basta questa richiesta? Se ad esempio `dimensione` fosse sempre uguale a `314`, il codice sarebbe corretto? Nonostante sia vero che tutte le cartelle abbiano una dimensione positiva o nulla, la porzione di invariante riportata non implica la correttezza dell'oggetto: bisogna richiedere che la dimensione sia la somma delle dimensioni delle voci contenute nella cartella `this`. Qual è la definizione di "invariante di rappresentazione"? È una funzione con che dominio? Con che codominio? In questo esempio, il dominio è il prodotto cartesiano tra l'intero `dimensione` e la lista `entries`; il codominio è un valore booleano. L'invariante di rappresentazione deve catturare la relazione che si instaura tra `dimensione` e `entries` affinché lo stato dell'oggetto sia coerente e corretto: non tutte le coppie `(dimensione, entries)` vanno bene. Consideriamo un altro pezzo di invariante:

[ ]:
```
/*
** REP INV: […]
**    entries != null
**    entries non può contenere elementi di tipo diverso da `Entry`
**    […]
*/
```

Consideriamo anche il seguente metodo:

[ ]:
```
public void aggiungi(Entry e) {
  if (!(e instanceof Entry))
    throw new IllegalArgumentException();

  entries.add(e);
  // […]
}
```

Una lista di `Entry` può contenere oggetti di tipo diverso da `Entry`? Ad un metodo che prende in input oggetti di tipo `Entry` può essere passato un parametro di tipo diverso? Cosa vuol dire che Java è un linguaggio fortemente tipato? `entries` può contenere oggetti di tipo o supertipo `Entry`.

[ ]:
```java
/**
 * OVERVIEW: Le istanze di questa classe rappresentano un elenco ordinato di
 *    nomi di voci, in cui solo l'ultma può essere un file.
 */
public class Path implements Iterable<Entry> {

  private final List<String> strings;

  /** […] */
  public Path(List<String> e) {
    Objects.requireNonNull(e);
    strings = e;
  }

  /** […] */

}
```

Nel costruttore `Path` stiamo effettivamente controllando quanto riportato nella `OVERVIEW` della classe? Come potremmo fare? (Domanda retorica: non si può) Il punto è che un percorso può essere valido o meno solo quando immerso in un filesystem: richiedere che il percorso sia in grado di determinare la propria correttezza indica un grado di *separation of concerns* non adeguato. Cosa fa il seguente metodo?

[ ]:
```java
/**
 * Post-condizioni: Restituisce il percorso corrispondente alla stringa `s`.
 *    Solleva `NullPointerException` se `s == null`.
 */
public static Path percorsoDaStringa(final String s) {
  try (Scanner s = new Scanner(s)) {
    try {
      List<String> strings = new ArrayList<>();

      if (s.next().equals(""))
        strings.add("");
      else {
        while (s.hasNext()) {
          final String nome = s.next();
          strings.add(nome);
        }
      }

      return new Path(strings);
    } catch (NoSuchElementException e)
```

```
      throw new IllegalArgumentException(e);
  }
}
```

C'è una circostanza in cui il metodo `next` di `Scanner` può restituire la stringa vuota? A cosa serve il costrutto `try-catch` interno? (Quello che cattura `NoSuchElementException`? Perché controllare a parte il caso in cui `next` restituisce una stringa vuota? Osserviamo inoltre che, se `next` non restituisce la stringa vuota, la variabile `strings` perde il primo 'membro' letto da `line`. In ogni caso, `next` non restituisce mai una stringa vuota, ma una stringa senza spazi di lunghezza massima. Da questo punto di vista, sarebbe stato meglio iterare sulla stringa a mano. Ha senso rendere `FileSystem` un sottotipo di `Directory`? La funzione di astrazione ha lo stesso dominio dell'invariante di rappresentazione, dato da tutti i valori possibili per lo stato: posto che questi rispettano l'invariante, la funzione di astrazione definisce qual è l'oggetto astratto corrispondente. Il seguente metodo, che permette di spostarsi nel filesystem, è corretto?

[ ]:
```
/**
 * Post-condizioni: […]
 */
public void move(String s) {
  Objects.requireNonNull(s);

  for (Entry e : cartellaCorrente)
    if (e.nome().equals(s) && e instanceof Directory)
      cartellaCorrente = (Directory) e;
}
```

Cosa si intendere per "tipo apparente" e "tipo concreto"? Perché questi concetti sono assenti, per esempio, in C e presenti in Java? `List` è un tipo apparente o concreto? (Domanda retorica) `ArrayList` è un tipo apparente o concreto? Esempio in cui `ArrayList` è un tipo apparente. Se una variabile ha per tipo apparente `ArrayList`, cosa può avere per tipo concreto? Quale aspetto di Java rendere necessario parlare di tipi apparenti e concreti? Uno è il fatto che Java è un linguaggio tipato, ed i tipi sono tra di loro in una relazione gerarchica; qual è l'altro aspetto? Java supporta il polimorfismo: ad una variabile di tipo `T` è possibile assegnare una qualunque espressione il cui tipo è `S`, con `S` sottotipo di `T`. Il tipo apparente è quello che il compilatore desume a partire dalla dichiarazione delle variabili (e quindi per induzione sulle espressioni). La *type safety* del linguaggio garantisce che i tipo concreti siano sempre sottotipi dei tipi apparenti. —

La seguente `OVERVIEW` è ragionevole? Corrisponde a quanto abbiamo definito essere un'astrazione?

[ ]:
```
/**
 * OVERVIEW: Questa classe è un sottotipo di `Entry`, con un attributo in più e
 *    dei metodi per la gestione del nuovo attributo.
 *    Gli oggetti di questo tipo sono mutabili.
 */
public class Directory extends Entry {}
```

Il problema è che stiamo (parzialmente) discutendo di aspetti implementativi, e non delle caratteristiche e delle competenze associate all'entità in questione. L'astrazione per specificazione deve indicare cosa fa l'astrazione, non come fa. Che cos'è la funzione di astrazione? Qual è il suo do-

minio? Qual è il suo codominio? Nell'esempio di `Directory` (il cui stato è dato da una stringa ed una lista), il dominio è dato dal prodotto cartesiano tra tutti i valori che possono essere assegnati ad una stringa e tutti i valori che possono essere assegnati ad `List<Entry>`. Ad ogni coppia ottenuta tramite questo prodotto cartesiano, la funzione di astrazione deve associare l'astrazione `Directory`. Perché il seguente metodo non è il massimo?

[ ]:
```
/**
 * Post-condizioni: […]
 */
public List<Entry> contenuto() {
  return entries; // `entries` è una porzione dello stato di `Directory`
}
```

Il problema è che stiamo esponendo parte della rappresentazione. Se qualcuno modificasse `entries` dopo aver invocato tale metodo, potrebbe rompere l'invariante di rappresentazione? (Che è il seguente)

[ ]:
```
/*
** REP INV: super()
**    list != null
**    Gli elementi in `list` diversi da `null`
*/
```

Sì, basta invocare `entries.add(null)`. Fatto questo, l'invocazione del metodo `dimensione` solleva `NullPointerException`. L'invariante di rappresentazione non riguarda `dimensione`, perché questo non è un attributo della classe `Directory`; tuttavia è una sua competenza. A cosa serve estendere un tipo? Come si manifesta la differenza tra `IntSet` (genitore) e `SortedIntSet` (figlio)? Nel senso, come me ne accorgo dai suoi comportamenti? Se l'unica differenza è nell'implementazione (`IntSet` usa `List` mentre `SortedIntSet` usa `SortedList`), `IntSet` e `SortedIntSet` collassano nello stesso tipo. In che senso il figlio specializza i comportamenti del padre? Per esempio, l'iteratore di `IntSet` restituisce i suoi elementi in ordine arbitrario; viceversa, quello di `SortedIntSet` li restituisce in ordine crescente. In quale esempio potremmo avere una classe che ne estende un'altra raffinandone l'implementazione? La specifica resta uguale, ma internamente ottengo qualcosa di diverso. Meccanismo del dispatching: cos'è? Quando entra in gioco? Da quali fasi è caratterizzato? Il dispatching avviene anche quando il compilatore sa quale metodo invocare: semplicemente, la tabella di dispatching contiene un'unica voce; viceversa, in presenza di ambiguità, la tabella di dispatching viene popolata con i possibili metodi (in base anche ai tipi concreti). Il dispatching è un meccanismo fondamentale con cui tutti i linguaggi compilati svolgono le chiamate a funzione: per esempio, se nel sorgente è presenta l'invocazione del metodo `m`, il compilatore inserisce nella tabella il metodo `m` di `Object`, quello di `Number` e quello di `Integer`; durante l'esecuzione, il metodo corretto viene invocato sulla base del tipo concreto. Astrazione iterazione: cos'è? A cosa serve? Le domande sono sui primi nove capitoli. Di ciascun argomento è richiesta: - una definizione (plausibile); - un'introduzione al concetto nel contesto del corso; - scopo dell'argomento; - qualche esempio; - qualche dettaglio.

Evitare i concetti vaghi. Il lessico deve essere preciso, circostanziato e riflettere il più possibile quello del testo. —

Il candidato ha scelto di implementare una classe concreta `Entry`: che senso ha istanziare tale classe? Inoltre, questa non ha un metodo per reperirne la dimensione: come faccio ad ottenerla, per esempio

quando più sottotipi di `Entry` sono mescolati all'interno di una collezione? Sono costretto ad usare `instanceof`: questo approccio espone a qualche criticità? Cosa succederebbe se volessimo implementare anche, ad esempio, link simbolici o altri tipi di `Entry` con dimensione? Dovremmo riempire il metodo per calcolare la dimensione di `instanceof`, il che viola completamente il proposito della programmazione ad oggetti (cioè la costruzione di una gerarchia con una competenza condivida, che ciascun oggetto implementa nel modo che ritiene più opportuno). C'è bisogno di `instanceof` nel seguente metodo? Non avviene automaticamente l'invocazione del metodo corretto?

[ ]:
```java
@Override public String toString() {
  StringBuilder sb = new StringBuilder();

  for (Entry e : listOfEntry) {
    if (e instanceof File)
      sb.append(((File) e).toString());
    else
      sb.append(e.toString() + "*");

    sb.append("\n");
  }

  return sb.toString();
}
```

È opportuno che `Directory` non sia in grado di esplicitare la propria dimensione? No, visto che dovrebbe essere una sua competenza. Domanda da *Programmazione I*: se voglio ottenere una lista di stringhe separando una singola stringa in corrispondenza dei caratteri `:` che contiene, quante stringhe posso aspettarmi di ottenere? È necessario verificare le pre-condizioni di un metodo privato? `getArray` è un nome appropriato per un metodo della classe `Path`? Come ribadito più volte, non riuscire a trovare il nome per un metodo sta probabilmente ad indicare una inadeguatezza di tale metodo. Come invariante di rappresentazione della classe `Path` basta richiedere che la stringa `path` sia diversa da `null`? È una competenza di `Path` sapere se il percorso `this` esiste all'interno del filesystem? Cosa si intende per "astrazione per specificazione"? Dispatching: cos'è? Come funziona? I tipi apparenti non dipendono da `javac`, e neanche dal processo di compilazione. Java è un linguaggio fortemente tipato, quindi la dichiarazione di una variabile è sempre accompagnata dal suo tipo: quello è il tipo apparente della variabile. Per via del meccanismo di compilazione, il compilatore non è in grado di determinare il tipo concreto di una variabile ma unicamente il tipo apparente.

[ ]: