mapio – Orali del 21.01.2021

January 30, 2021

Volendo fornire un oggetto di tipo `Directory` della possibilità di ordinarsi, dove sarebbe più opportuno inserire questa competenza? Un filesystem dovrebbe al massimo essere in grado di elencare il contenuto di una directory, per cui il posto più adatto sarebbe la classe `Shell` (in particolare il comando `ls`). La classe `Entry` non implementa `equals`, mentre la classe `File` specifica che due file sono uguali se hanno lo stesso nome e dimensione; la classe `Directory`, viceversa, suppone due directories uguali se hanno lo stesso nome: qual è il problema in tutto ciò? Stiamo rendendo i due sottotipi impossibili da confrontare, il che sarebbe normale se `equals` avrebbe la tipica semantica. Ma se usassimo `equals` per controllare la presenza di duplicati in una directory, un'implementazione come quella descritta sopra non raggiunge lo scopo desiderato. (Quindi secondo il professore un file ed una directory con lo stesso nome non possono stare nella stessa directory?) Una soluzione migliore da questo punto di vista sarebbe stato implementare `equals` solo in `Entry`, specificando che chi la estende non può sovrascrivere tale metodo (esempio di specifica in funzione dell'estendibilità).

Quali tecniche di astrazione abbiamo visto? Cosa intendiamo per "astrazione procedurale"? Che caratteristiche deve avere tale astrazione? Perché è utile avere uno strumento del genere? Esempio pratico di funzione parziale che **non conviene** definire totale, o rendere totale attraverso l'uso di eccezioni (ricerca binaria). Esempio pratico di funzione parziale che **conviene** rendere totale: calcolo della radice quadrata su numeri negativi. In caso di funzione parziale, quale tipo di eccezione è meglio utilizzare per renderla totale? Perché? `Try-catch` è l'unico modo che abbiamo per gestire un'eccezione controllata? L'uso di un'eccezione non controllata nel metodo `a` è plausibile se introduciamo nella classe un metodo `b` per verificare che l'input di `a` non sollevi un'eccezione. Se decidessimo di rendere totale una funzione introducendo un'eccezione, dovremmo quindi aggiungere un metodo che 'illustri' al cliente quando tale eccezione viene sollevata: nell'esempio della ricerca binaria, potremmo quindi definire un metodo `isSorted` che verifica se il vettore è ordinato. Gli errori associati al filesystem sono eccezioni controllate, nonostante esistano metodi per verificare l'esistenza di un file: come mai? Programmazione concorrente: tra l'invocazione del metodo che verifica l'esistenza di un file `f` ed un altro metodo che, per esempio, apre `f`, potrebbe inserirsi un ulteriore thread che elimini `f`. Cfr: TOCTOU (Sicurezza e Privatezza) Supponiamo di avere un tipo `T` con un certo metodo, ed un sottotipo `S` di `T` che sovrascrive tale metodo: quale relazione esiste tra la specifica del metodo nel supertipo `T` e nel sottotipo `S`? Il comportamento dei due metodi deve essere simile: a quale principio è legato questo aspetto? Principio di sostituzione. Esempio pratico di metodo sovrascritto, in cui rilassiamo le pre-condizioni e rafforziamo le post-condizioni. (Proposto dal candidato) Il genitore ha un metodo che calcola la radice quadrata di numeri strettamente positivi, mentre il figlio di numeri anche nulli. E se il figlio restituisse `-3` quando riceve in input `0`? Questo comportamento rispetta il principio di sostituzione? Stiamo rilassando le pre-condizioni e rafforzando le post-condizioni correttamente? Il metodo del genitore non specifica cosa accade se l'input è `0`, per cui non possiamo dire che restituire `-3` quando l'input è `0` sia un comportamento del figlio errato. Infatti, il comportamento del figlio deve essere lo stesso del padre a patto che le pre-condizioni del padre siano verificate. —

Perché l'attributo `nome` della classe `Entry` è protetto? Senza documentare questa scelta, una sottoclasse potrebbe rompere l'invariante di rappresentazione. Una scelta più ragionevole sarebbe utilizzare `public` e `final` per `nome`, tanto è un attributo immutabile e difficilmente un nome può essere rappresentato in modi diversi. Perché gli oggetti di tipo `File` sono immutabili, nonostante l'attributo `dimensione` non sia `final`? Perché `dimensione` è `private` e non vi sono metodi di mutazione. Meccanismi di sovraccaricamento e sovrascrittura in Java: cosa sono? L'esempio più semplice di sovraccaricamento è quando avviene in una singola classe, per cui ho almeno due metodi con lo stesso nome e parametri diversi in numero o tipo. Se invece ho due metodi con la stessa segnatura in classi diverse della gerarchia, si parla di sovrascrittura. È possibile che queste due 'realtà' si mescolino in qualche modo? Può avvenire il sovraccaricamento tra classi diverse? Se sì, sarebbe una buona cosa? Per esempio, nella classe figlio abbiamo un metodo `subset` ereditato e sovrascritto, oltre ad un metodo `subset` che sovraccarica il precedente: in questo caso il compilatore si comporta esattamente nel modo in cui il programmatore si aspetta, andando ad invocare il metodo più specifico. C'è però una circostanza più preoccupante, in cui il programmatore sovraccarica un metodo anziché sovrascriverlo; un esempio eclatante è quello di `equals`: basta infatti cambiarne la segnatura (tipo `equals(Tipo t)` anziché `equals(Object obj)`) affinché il meccanismo di dispatching dinamico non funzioni più. Abbiamo la seguente porzione di codice:

[5]:
```
public class Appropriate {

  static class WhichOne {
    public void method(int x) {
      System.out.println("WhichOne::method");
    }
  }

  static class WhichOneSubtype extends WhichOne {
    public void method(double x) {
      System.out.println("WhichOneSubtype::method");
    }
  }

  public static void main(String[] args) {

    WhichOne wo = new WhichOne();
    WhichOne woa = new WhichOneSubtype();
    WhichOneSubtype wos = new WhichOneSubtype();

    wo.method(1);
    woa.method(1);
    wos.method(1);

  }

}
```

Cosa viene stampato? `wo.method(1)` stampa `WhichOne::method`, ma come mai anche `wos.method(1)` stampa la stessa cosa e non `WhichOneSubtype::method`? Il fatto è che la classe

`WhichOneSubtype` non sta sovrascrivendo il metodo `method`, ma è come se l'avessimo scritta così:

[ ]:
```
static class WhichOneSubtype extends WhichOne {
  public void method(int x) {
    System.out.println("WhichOne::method");
  }

  public void method(double x) {
    System.out.println("WhichOneSubtype::method");
  }
}
```

Quindi, invocando `method(1)`, il metodo più specifico (cioè che richiede meno conversioni) è quello con segnatura `method(int)`. Cosa sono l'invariante di rappresentazione e la funzione di astrazione? A cosa servono? Cos'è una funzione iniettiva? Associa elementi diversi del codominio ad elementi diversi del dominio. La cosa interessante della funzione di astrazione è che dovrebbe essere suriettiva, per cui ogni punto del codominio ha almeno un punto corrispondente nel dominio: per esempio, ciascun oggetto astratto 'Insieme' ha almeno un vettore (se questa è la rappresentazione scelta) associato. Perché la funzione di astrazione deve necessariamente essere suriettiva, e sarebbe un errore altrimenti? Non avrebbe senso scegliere una rappresentazione che non mi consente di rappresentare tutto quanto è legato all'astrazione in esame. Cosa intende la Liskov per *fully populated*? Devono essere messi a disposizione metodi adeguati, affinché con un'opportuna sequenza di invocazioni possa ottenere tutti i possibili stati dell'astrazione. Supponiamo di implementare un insieme tramite vettore, come fa la Liskov. Abbiamo visto che l'invariante di rappresentazione esclude la presenza di duplicati nell'array: sarebbe sensato, viceversa, implementare l'astrazione permettendo duplicati all'interno della rappresentazione? Se sì, come cambierebbe l'invariante di rappresentazione e la funzione di astrazione? L'implementazione di quali metodi risulta essere più semplice, e di quali più complicata se escludo la presenza di duplicati nella rappresentazione? La rappresentazione è libera: potrei scegliere di rappresentare un insieme con una stringa, nonostante sia complicato lavorarci. In base alla rappresentazione, si scelgono l'invariante di rappresentazione e la funzione di astrazione. In quale fase di progetto diventano importanti tali funzioni? Per verificarne la correttezza. Come usiamo a questo scopo la funzione di astrazione? La usiamo sia direttamente (dalla rappresentazione all'astrazione) che nel modo inverso (dall'astrazione alla rappresentazione): qui diventa importante l'iniettività, poiché fissata un'astrazione posso avere più rappresentazioni. Se avessi una funzione di astrazione anche iniettiva, le dimostrazioni di correttezza sarebbero semplici. A cosa servono gli iteratori? La classe che rappresenta un iteratore viene inserita all'interno della classe da iterare, perché così ho accesso diretto alla rappresentazione. Cosa cambierebbe tra avere una classe statica interna ed una classe statica esterna? I modificatori di visibilità: se la classe statica si trova all'esterno della classe da iterare, non ho modo di accedere ai suoi attributi privati. —

La classe `FolderEntry` non fornisce alcun metodo per aggiungere una voce alla cartella `this`, rendendo di fatto tale classe immutabile: è una buona scelta? Cos'è un `Builder`? Se le cartelle fossero immutabili, per aggiungere un elemento alla radice dovrei effettuare un'intera copia del filesystem: poco plausibile. Implementare in maniera immutabile un'entità che per sua natura è mutabile è una scelta che bisogna giustificare. In questo caso, `FolderEntry` è solo un avvolgitore di `HashSet` (la rappresentazione scelta dal candidato). `Filesystem` è la classe ragionevole in cui tenere traccia della cartella corrente? Posto che tale scelta sia corretta, il nome `Filesystem` non è adatto ad una classe del genere: quale altro nome potremmo darle? (Domanda retorica) Non sapere che nome

dare ad una classe è un importante indice per capire che sono state fatte scelte discutibili circa la suddivisione delle competenze. Quale classe mantiene la collezione di `Entry` e della loro organizzazione gerarchica? Quali competenze ha tale classe? Cosa si intende per 'comando'? (Domanda retorica) Un file system deve essere in grado di aggiungere una voce in una cartella: supponendo di implementare questa competenza con un metodo, di che tipo sono gli argomenti in input? La scelta più adeguata sarebbe avere un percorso ed una voce (il primo argomento rappresenta la posizione in cui aggiungere la voce). Potrebbe aver senso fornire un metodo per la rimozione di una voce: quali argomenti prende in input? Inserire la cartella corrente nel filesystem significa che, se avvio due shell spostandomi in cartelle diverse, ho due filesystem (il che chiaramente non è vero per alcun sistema operativo ragionevole). Cosa si intende per 'mutabilità', 'immutabilità', "condivisione dello stato"? Perché sono concetti fondamentali? Non è detto che un oggetto sia mutabile perché posso accedere ai suoi campi: è una condizione sufficiente ma non necessaria. Qual è la definizione di "oggetto mutabile"? Un oggetto è mutabile se il suo stato può essere modificato: come questo avvenga è irrilevante. Mutabilità ed immutabilità sono caratteristiche dell'implementazione o della specifica? Se la specifica fosse immutabile e l'implementazione mutabile, non starei facendo un errore? Cosa vuol dire che «una libreria espone un comportamento?» (Domanda retorica) La proprietà di linguaggio è cruciale. La mutabilità è una caratteristica della specificazione: non avrebbe senso dover leggere il sorgente per capire se una classe è mutabile o meno. Quando definisco un tipo di dato astratto, devo dire se è mutabile o immutabile: nel primo caso, devo fornire metodi di mutazione (che permettano di modificare lo stato dell'oggetto); nel secondo, garantisco che il suo stato non cambi dopo la costruzione (quindi nelle specifiche non posso aggiungere alcun metodo mutazionale). Detto questo, può anche succedere che l'implementazione di un oggetto immutabile faccia uso di astrazioni mutabili, dovendo comunque garantire che: - lo stato di quest'ultime non cambi, oppure - i cambiamenti di stato non sono percepibili all'esterno

Se la specifica richiede che l'astrazione sia immutabile, l'implementazione deve essere nel complesso immutabile: per esempio, la Liskov rende i polinomi immutabili ma li implementa tramite array, che sono strutture dati mutabili. Non è necessario che la rappresentazione sia immutabile affinché anche l'oggetto lo sia; non posso scegliere di implementare in modo mutabile un oggetto immutabile, ma posso scegliere di avere una rappresentazione basata su oggetti mutabili. Supponiamo di avere un oggetto dallo stato immutabile, ma l'oggetto in sé è mutabile: come mai potrebbe accadere una cosa del genere? In che senso «un costruttore definisce un comportamento»? (Domanda retorica) Ad esempio, abbiamo un oggetto che come unico attributo ha un intero `final`: tale oggetto è necessariamente immutabile? Una rappresentazione immutabile garantisce che l'oggetto sia immutabile? Posso determinare la mutabilità di un oggetto unicamente in base al codice sorgente? (Dovrebbe esserci un esempio in Effective Java) È necessario osservare la specificazione di un oggetto per stabilire se questo è immutabile o meno: se anche un singolo metodo può restituire valori diversi, per esempio in base alla data e l'ora, il cliente può osservare dall'esterno due stati diversi dell'oggetto nonostante la sua rappresentazione sia immutabile. Un esempio di questo genere si trova nelle API di Java, in particolare nella classe che implementa gli URL (cfr. Effective Java). Perché queste caratteristiche sono rilevanti nell'ambito della condivisione? Cosa si intende per "condivisione di oggetti"? `String p = "pippo"` È pericoloso condividere `p`? No, è pericoloso condividere l'oggetto a cui `p` si riferisce. Si parla di "condivisione di oggetti" quando a due variabili distinte viene assegnato il riferimento al medesimo oggetto. A quale problema può portare la condivisione nel caso di oggetti mutabili? È possibile invocare un metodo accedendo all'oggetto tramite una delle variabili, e perché questo è un problema? Facciamo un esempio concreto. Quali sono le caratteristiche di Java che lo rendono un linguaggio sicuro dal punto di vista dei tipi (*type safe*)? Astrazione procedurale: cosa si intende per "funzione totale" o 'parziale'? Quando le adoperiamo? Quali considerazioni derivano

da questa suddivisione? Su quale astrazione si basa quella procedurale? Sulla parametrica e su quella per specificazione: la prima astrae dai parametri effettivi, la seconda dall'implementazione concreta indicando il comportamento di un metodo. Una funzione è totale quando la sua clausola 'pre-condizioni' è vuota; è parziale altrimenti: in tal caso, la specificazione sta restringendo l'insieme di valori possibili per i suoi input. Perché usiamo funzioni di un tipo e dell'altro? È per una singola funzione che posso decidere se renderla totale o parziale. Potrei ad esempio avere una funzione che effettua la ricerca dicotomica, e renderla totale. Cosa significa «restituire un *throws*»? (Domanda retorica) Le funzioni parziali possono essere rese totali controllando che gli input specificati rispettino la clausola `pre-condizioni`, e sollevando un'eccezione altrimenti; questo non è sempre opportuno (esempio della ricerca binaria). È sempre possibile rendere totale una funzione parziale? In altri termini, la clausola `pre-condizioni` può fare solo richieste che è possibile verificare automaticamente? Esiste un risultato fondamentale dell'informatica, il quale indica come non tutte le funzioni sono computabili (problemi non decidibili): ad esempio, se avessi un metodo `m` che prende in input un altro metodo `e` ed un argomento `a`, e la pre-condizione fosse «il metodo `e` deve terminare su input `a`», non ho modo di rendere totale il metodo `m` (problema dell'arresto). Quindi non si può sempre rendere totale una funzione parziale, perché la clausola `pre-condizioni` potrebbe contenere un predicato che non si può verificare automaticamente. Quando adoperiamo le eccezioni per rendere totale una funzione parziale, quali scelte abbiamo? In base a cosa ci conviene scegliere tra controllate e non controllate? In presenza di un'eccezione controllata, il compilatore segnala errore se manca il costrutto `try-catch`? Non necessariamente, perché è sempre possibile 'riflettere' l'eccezione ad un livello più alto di astrazione. Per scegliere tra i due tipi di eccezione, devo pensare attentamente alle circostanze in cui mi aspetto che il metodo venga chiamato: se l'eccezione è controllata, il cliente deve necessariamente gestirla in qualche modo; se è non controllata, il mio obbiettivo è quello di interrompere l'esecuzione di codice che potrebbe non essere sicuro, segnalando un malfunzionamento. —

Che senso ha fornire `Entry` di un *getter* astratto per l'attributo `nome`? Può avere senso un *getter* anche se il campo è pubblico, nonostante sia piuttosto inutile. Quando si adoperano i metodi astratti? Nel nostro caso sia `File` che `Directory` implementano il *getter* astratto restituendo il nome, e causando quindi una duplicazione di codice. Ci sarebbe potuto essere un senso se le classi concrete avessero restituito il nome 'decorato' come richiesto dai requisiti (anche se una competenza del genere dovrebbe essere della shell). Perché `equals` e `hashCode` sono stati inseriti nella classe `Directory`? Qual è il loro scopo? Bisogna valutare attentamente la possibilità di implementare `equals`, in particolare per oggetti mutabili come `Directory`. L'implementazione di `equals` non verifica l'uguaglianza di tutto lo stato, ma solo di `name` e `weight`: è una scelta corretta? Supponiamo di avere due file con lo stesso nome, la stessa dimensione, lo stesso contenuto: i due file sono uguali dal punto di vista del filesystem? No, mi interessa anche il loro percorso. Quindi, in questo caso, l'implementazione di `equals` va specificata (nonostante stessimo sovrascrivendo un metodo di `Object`) perché non rappresenta la nozione naturale di uguaglianza: in particolare bisogna motivare la scelta di considerare uguali due cartelle se hanno lo stesso nome e la stessa dimensione. Perché `weight` (nella classe `Directory`) è un attributo, e non una variabile locale? Quali vantaggi e svantaggi hanno queste due possibilità? Essendo parte dello stato, `weight` deve figurare nell'invariante di rappresentazione e nella funzione di astrazione; inoltre, la sua correttezza deve essere verificata in ogni metodo mutazionale, e può incidere sui metodi che lo utilizzano senza necessariamente modificarlo: tanto lavoro per un beneficio pressoché nullo. Anche effettuare il 'caching' di questo attributo sarebbe una scelta critica, perché una cartella dovrebbe notificare il genitore di essere stata modificata (in modo che quest'ultimo possa ricalcolare il suo peso). Ha senso che il metodo `add` aggiunga la voce alla cartella senza fare alcun controllo (a parte quello di nullità)? Quali altri

controlli avrei dovuto fare? Nella stessa cartella non dovrebbero essere due voci con lo stesso nome. Se implementassi questa riflessione tramite il metodo `contains`, funzionerebbe? Qual è la specifica di `contains` (i.e. quando un oggetto è contenuto in una collezione)? Utilizza `equals` in qualche modo. Quindi, tenendo in considerazione **dove** abbiamo implementato `equals`, invocare `contains` può funzionare? Sarebbe stato meglio inserire `equals` nella classe `Entry` (quindi ovviamente non avrei dovuto considerare la dimensione): perché? Quando c'è di mezzo una gerarchia, `equals` diventa complicato poiché è facile invalidare alcune delle proprietà associate al suo contratto (per esempio la simmetria). Consideriamo la seguente riga:

[ ]:
```
if (!(d instanceof Directory))
```

Cosa succede se `d` fosse un'istanza della classe `File`? Per esempio:

[ ]:
```
Directory root = new Directory("");
File f = new File("pippo", 10);
root.add(f);

Directory d = new Directory("pippo");
root.add(d);
```

Cosa succede? Cosa vorrei succedesse? Se avessi due voci dallo stesso nome in una cartella, e l'utente mi chiedesse di cancellare una delle due, quale cancello? (Domanda retorica) Perché invece l'implementazione di `equals` proposta permette di avere nella stessa cartella una cartella ed un file dallo stesso nome? Cosa stampa la seguente riga?

[ ]:
```
System.out.println(f.equals(d));
```

Perché dice `false`? Il problema è che il nostro `equals` controlla l'uguaglianza solo tra oggetti omogenei (`File` con `File`, `Directory` con `Directory`). Potrei spostare questa competenza in alto nella gerarchia, oppure utilizzare:

[ ]:
```
if (!(d instanceof Entry))
```

In alternativa, piuttosto che implementare `equals` e `hashCode`, sarebbe sufficiente controllare manualmente che la voce passata al metodo `add` abbia un nome diverso da tutte quelle presenti nella cartella `this`. Le scelte di non implementare una classe che rappresenti un percorso, e di inserire il percorso della cartella corrente tra i campi della classe `Filesystem` (sotto forma di `String`) sono plausibili? Chi dovrebbe occuparsi del concetto di "cartella corrente", la shell o il filesystem? La shell. Cosa fa il seguente metodo? (Domanda retorica)

[ ]:
```
public void mkdir(String name, String path, Directory currentDirectory) {
  if (path.equals(""))
    currentDirectory.add(new Directory(name))
  else {
    Directory d = getDirectory(this.path += path);
    d.add(new Directory(name))
  }
}
```

In particolare, come si comporta se lo invoco in questo modo?

[ ]:
```
mkdir("pippo", "", new Directory());
```

Crea un cartella di nome `pippo`, la inserisce in una nuova cartella appena generata (chiamiamola `n`) e, quando il controllo torna al chiamante, interviene il *garbage collector* eliminando la cartella `pippo` ed `n` dallo heap. Questo metodo, di fatto, non fa nulla di utile. Il seguente metodo funziona sempre?

[ ]:
```
/**
 * Post-condizioni: Restituisce la cartella presente al percorso `path`.
 */
private Directory getDirectory(String p) {
  Directory d;
  String[] parts = p.split(":");

  d = root;

  for (int i = 0; i < parts.length; i++)
    d = d.subDirectory(parts[i]);   * @return


  return d;
}
```

Anche se `p == null`, il metodo fallisce quasi subito quindi il problema principale non è quello. Che succede se `p` è il percorso di un file? Astrazione iterazione in generale. Quali possibilità abbiamo per implementarla? Quale sarebbe il problema di esporre la rappresentazione? Uno secondario è che il cliente potrebbe basarsi su di essa per effettuare le sue operazioni, e quindi in un secondo momento non potrei cambiare l'implementazione della classe. Talvolta esporre la rappresentazione non è una buona scelta, ma il programmatore potrebbe anche valutare che lo sia (dipende da caso a caso); allo stesso modo, non è detto che utilizzare gli iteratori non esponga la rappresentazione (dipende da come l'iteratore viene scritto). Cosa si intende con l'espressione «Java è un linguaggio *type safe*»? Cosa implica concretamente la presenza di *automatic storage management*? In Java può accadere che un oggetto allocato nello heap venga deallocato? Cosa fa il *garbage collector*? Durante la fase di compilazione, il compilatore è in grado di individuare gli errori sui tipi impedendo la generazione dell'eseguibile: stiamo parlando di tutti gli errori di tipo, o solo alcuni? (Che quindi dovrebbero essere controllati in un secondo momento, per garantire la *type safety*) Il compilatore determina il tipo apparente delle espressioni a sinistra dei punti (nell'invocazione di un metodo), il tipo apparente dei parametri e verifica che nella gerarchia ci sia un metodo che è possibile invocare. Fa qualche altro controllo? Ce n'è uno più banale, legato all'esistenza di tipi apparenti e concreti. Verifica anche, se assegnamo un'espressione di tipo apparente `b` ad una variabile di tipo apparente `a`, `b` sia un sottotipo di `a`. Durante l'esecuzione non possono sorgere altri problemi con i tipi? Se ho una variabile di tipo `Entry`, ma sono convinto che il suo tipo concreto sia `Directory`, cosa faccio per invocare un metodo della classe `Directory`? Uso il casting, trasformando un tipo più generale in uno più specifico. Il compilatore non ha modo di sapere se questa operazione va a buon fine durante l'esecuzione, e infatti `ClassCastException` è una `RuntimeException`. Assegnamento agli array (non ho capito cosa intendesse qui) Cosa sono le eccezioni? A cosa servono? (Sia a livello teorico che pratico) Se l'eccezione è non controllata, devo segnalarla nella segnatura del metodo ed avvolgerne l'invocazione in un `try-catch`? Posso scegliere, purché io le gestista: ho la possibilità

di avvolgere l'invocazione del metodo in un `try-catch`, oppure di indicarle nella sua segnatura (a suggerire che se ne deve occupare chi invoca il metodo che sto scrivendo). In quale circostanza può essere utile catturare un'eccezione per poi ignorarla? —

Che senso ha rendere il metodo `toString` astratto nella classe `Entry`? Questa scelta ha conseguenze sintattiche? Sì, tutte le classi che estendono `Entry` sono costrette ad implementare `toString`. Sarebbe quindi meglio evitare una cosa del genere, a meno che non voglia 'aumentare' la specifica del metodo in questione: potrei infatti richiedere come le classi concrete implementino `toString`. L'invariante di rappresentazione della classe `Directory` è corretto?

[ ]:
```
/*
** REP INV: RI(super)
**     contenuto != null
**     ogni voce in `contenuto` diversa da `null`
**     dimensione >= 0
*/
```

Bisognerebbe specificare che l'attributo `dimensione` deve coincidere con la somma delle dimensioni delle voci contenute in `contenuto`. Come fanno i metodi `add`, `remove`, `contains` di `HashSet<Entry>` a funzionare se `Entry` non implementa `equals`? Cosa si intende per 'duplicato'? (Domanda retorica) Verosimilmente, `HashSet` utilizza `equals` per verificare se un certo elemento è contenuto nell'insieme; senza sovrascrivere `equals`, l'insieme utilizza quello della classe `Object`. Nel caso voglia implementare `equals`, devo documentare attentamente quando due `Entry` sono uguali (perché, per evitare duplicati, basta che due voci abbiano lo stesso nome per essere uguali, indipendentemente dalla loro dimensione). Ha senso effettuare un *caching* della dimensione nel seguente modo?

[ ]:
```
public void aggiungiEntry(Entry e) {
  Objects.requireNonNull(e);

  boolean aggiunto = contenuto.add(e);
  if (aggiunto)
    dimensione += e.dimensione();
}
```

Se creo un file in una cartella contenuta in `this`, quest'ultima non ha modo di saperlo e non aggiorna correttamente la sua dimensione. Attenzione al *caching*: meglio partire dalla soluzione ovvia in fase di esame. La classe `AbsolutePath` ha solamente un costruttore ed il metodo `toString`: è quello che vogliamo? Qualunque sia l'astrazione che stiamo costruendo, l'assenza di un metodo osservazione (oltre a `toString`) deve essere un campanello d'allarme. Quali altre competenze potrebbe avere un percorso? Ha senso avere una classe `AbsolutePath` ed una classe `RelativePath`? Cosa cambierebbe tra le due? Dovrebbero essere implementati alcuni metodi per aggiungere un percorso relativo ad uno assoluto. L'insieme di competenze dovrebbe quindi essere: - costruire un percorso a partire da una stringa (costruttore, o metodo di fabbricazione); - 'montare' (e 'smontare') percorsi; - costruire una stringa a partire da un percorso (`toString`).

Si potrebbe quindi pensare di avere un campo `boolean` che distingua percorsi relativi ed assoluti. Anche nel caso di `Entry`, può avere senso un metodo che restituisca `true` se `this` è un'istanza di `Directory` e `false` altrimenti. Astrazione iterazione: cos'è? A cosa serve? Quali tecniche mette a disposizione Java per implementarla? Cosa si intende per "iterazione interna"? Perché

non sarebbe 'opportuna?' (Domanda retorica) Se non riesco a fornire un meccanismo di iterazione interna, ho due modi per implementare l'iterazione esterna: fornire accesso alla rappresentazione (che potrebbe non essere un'ottima scelta), oppure un iteratore. Quali benefici offre l'astrazione per specificazione? La Liskov parla di località e mutabilità: cosa sono? L'isolamento consentito dalla specifica rispetto all'implementazione di un'astrazione è tale per cui, se decido di modificarla, non devo toccare niente di quanto le "sta attorno:" infatti, i clienti dell'astrazione in esame si basano unicamente sulla sua specificazione per utilizzarla. Viceversa, senza astrazione per specificazione, i clienti dovrebbero fare affidamento alla sola implementazione: quindi o questa rimane sempre uguale, oppure devo accertarmi che eventuali modifiche non rompano i clienti. Allo stesso modo, la località mi permette di apportare miglioramenti all'implementazione senza sapere come i clienti utilizzano l'astrazione associata: devo unicamente soddisfare un insieme di richieste locali alla mia classe. —

Astrazione iterazione. "Iterazione interna" significa che, se ad esempio voglio calcolare la somma degli elementi di un insieme, inserisco nella classe `Insieme` un metodo che lo faccia. Una possibilità per l'astrazione iterazione è esporre la rappresentazione, permettendo ai clienti di iterare su una sua copia: gli svantaggi di questo approccio sono una scarsa efficienza e l'assenza di località (se volessi cambiare la rappresentazione, dovrei tener conto del codice dei clienti). Quali meccanismi sintattici permettono di realizzare un iteratore? Le classi anonime permettono di non dare un nome alla classe, e sono utili se la voglio utilizzare una sola volta. In quale altre occasioni le abbiamo utilizzate? Spesso con i comparatori. Meccanismo di dispatching: come avviene l'invocazione di un metodo? Quali passi devono essere seguiti in compilazione ed esecuzione? Che intendiamo dicendo che il compilatore 'cerca' il metodo nella classe del tipo apparente o nei suoi supertipi? Su cosa si basa la ricerca? Sul tipo apparente, sul nome del metodo e sulla segnatura. C'è un caso in cui tale ricerca può fallire? Sì, per esempio quando due metodi sovraccaricati richiedono lo stesso numero di conversioni per gli argomenti in input: ciò accade durante la compilazione (che chiaramente fallisce), per cui basandosi unicamente sui tipi apparenti; è possibile che, se il compilatore avesse prodotto il sorgente, durante l'esecuzione non ci sarebbero stati problemi (visto che la JVM lavora sui tipi concreti). Perché invece questo aspetto va risolto in compilazione? Il compilatore vuole dirci che, qualunque cosa succeda, il metodo in questione può essere chiamato: se invece rimandasse il problema a tempo di esecuzione, ci sarebbero alcuni casi in cui può disambiguare ed altri in cui non può. Il casting permette di cambiare il tipo apparente di un riferimento: il compilatore lo permette ma, grazie alla proprietà di *type safety* la JVM interrompe immediatamente l'esecuzione se la conversione non va a buon fine. 'Recentemente,' alle interfacce di Java sono stati aggiunti i metodi di default: cosa sono? Perché non si sposano bene con le classi astratte? (Effective Java) Come funzionano i metodi di default se non hanno accesso allo stato dell'oggetto? (Che non è presente in un'interfaccia) Può invocare altri metodi presenti nell'interfaccia. Per cosa si potrebbe usare questa feature? Per esempio, stiamo costruendo un interfaccia `Insieme`, la quale prescrive la presenza di un iteratore: avrebbe senso costruire un metodo di default per il metodo `contiene`? Non è detto che l'iteratore esponga la rappresentazione: il generatore restituisce uno per uno gli elementi dell'insieme, su cui invoco `equals`. È la soluzione ottimale? Non è detto, dipende dalla rappresentazione delle classi che implementano l'interfaccia; ma intanto garantisco che tutte loro abbiano implementato un `contiene`. I metodi di default sono stati aggiunti perché, con l'introduzione degli stream, le collezioni avevano bisogno di capacità assenti nelle classi concrete: inserendole tramite metodi di default in un'interfaccia, non si rompe nessuna classe che la implementa; il programmatore resta comunque libero di fornire un'implementazione migliore rispetto a quella offerta dai metodi di default.

[ ]: