

Esercizio Composite:

un file system

Anche questa esercitazione ha lo stesso schema della volta scorsa. L'esercizio è suddiviso in diversi passi, da svolgere in sequenza. Inoltre, l'esercizio ha i seguenti obiettivi:

- Implementare il programma gradualmente, una classe alla volta, e passare alla successiva solo dopo aver testato la classe appena implementata. Per questo motivo, NON si deve implementare tutti i punti insieme e testarli solo alla fine.
- Familiarizzare con alcuni strumenti di refactoring.
- NON creare immediatamente astrazioni, ma implementare prima una classe concreta, testarla e poi tramite refactoring creare delle astrazioni (classi astratte e/o interfacce) per un possibile riuso futuro (cioè in vista del prossimo passo dell'esercizio).
- Familiarizzare con gli strumenti dell'IDE per essere produttivi durante l'implementazione, il testing e il refactoring.

Ricordate: abituatevi a rilanciare i test dopo ogni modifica (memorizzate le shortcut di Eclipse). Ricordate che i test andrebbero scritti in una directory sorgente separata, es., *tests*.

In questa esercitazione modelleremo un *file system*, composto quindi da *file* (oggetti foglia) e *directory* (oggetti composti), usando il pattern *Composite*. In particolare, applicheremo il pattern nella **forma type safe**. Quindi la gestione dei *children* sarà presente solo nel componente composto (in questo esempio, la *directory*). In questo modo, staticamente, non sarà possibile commettere l'errore di aggiungere un figlio in un componente foglia (in questo esempio, il *file*).

Ovviamente il file system sarà solo “modellato”: gli oggetti che creeremo (e le loro classi) non implementeranno effettivamente le funzionalità per scrivere e leggere da un file system vero e proprio. Cioè, non scriveremo, ne' leggeremo, direttamente file o directory. (In un'applicazione reale, la lettura da e scrittura sull'hard disk sarebbero implementate in un'altra parte dell'applicazione, che userebbe comunque la parte che modella il file system che implementiamo in questa esercitazione.)

Per evitare ambiguità con le classi di Java, come *File*, useremo sempre il prefisso *FileSystem* per le classi e tipi che creeremo, es. *FileSystemFile*, *FileSystemDirectory*, ecc. Nelle soluzioni useremo come nome di pacchetto *mp.exercise.filesystem*. Ovviamente potete scegliere un altro nome di pacchetto. Il nome del package verrà esplicitamente mostrato nello svolgimento solo quando è strettamente necessario.

Sempre per semplicità, non modelleremo altre informazioni tipiche in un file system, come data di modifica, dimensione del file, ecc, ma solo il nome.

Questa volta avremo una classe di test separata per ogni classe concreta che implementeremo, quindi *FileSystemFileTest* per *FileSystemFile*, ecc.

1 File

Implementare una classe *FileSystemFile* che rappresenta una *foglia* del pattern *Composite*. La classe ha un campo *name* e un metodo *void ls(FileSystemPrinter)* per “listare” il file usando un’interfaccia, che deve essere creata, *FileSystemPrinter* che contiene un solo metodo *void print(String)*. Ad esempio, quando si chiama il metodo *ls* su un file col nome “pippo”, la stringa passata al metodo *print* di *FileSystemPrinter* dovrebbe essere “File: pippo”. Per i test ci servirà un’implementazione di *FileSystemPrinter*. Che implementazione di tale interfaccia potremmo creare? E dove? Tenete conto che questa implementazione di *FileSystemPrinter* è al momento fittizia e ci serve solo per testare la classe *FileSystemPrinter*. Cioè, ci serve per esser sicuri che quando si chiama il metodo *ls* su un file, ad esempio, col nome “pippo”, la stringa passata al metodo *print* di *FileSystemPrinter* dovrebbe essere “File: pippo”. Notare che il metodo *ls* è void, quindi la fase di verifica non potrà controllare un valore di ritorno, ma dovrà controllare che si verifichino certi effetti collaterali sull’argomento passato (*FileSystemPrinter*).

Usando i tool di Eclipse, visti alla scorsa esercitazione, creiamo la classe, il costruttore e il campo.

```
package mp.exercise.filesystem;

public class FileSystemFile {

    private String name;

    public FileSystemFile(String name) {
        this.name = name;
    }

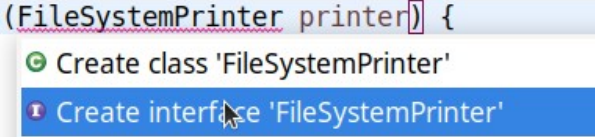
}
```

Creiamo il metodo pubblico *ls(FileSystemPrinter)* (la volta scorsa abbiamo visto un “template” di Eclipse per creare velocemente un metodo pubblico). Il codice non compilerà perché *FileSystemPrinter* non esiste ancora. Usare il Quickfix di Eclipse per creare l’interfaccia:

```
public FileSystemFile(String name) {
    this.name = name;
}

public void ls(FileSystemPrinter printer) {
}

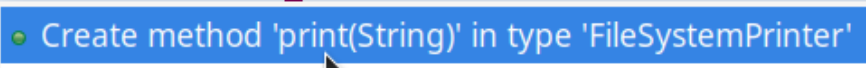
}
```

A screenshot of the Eclipse IDE showing a code editor with the `ls` method. A red squiggly line under the `FileSystemPrinter` parameter triggers a Quickfix menu. The menu has two options: "Create class 'FileSystemPrinter'" (highlighted in blue) and "Create interface 'FileSystemPrinter'" (highlighted in blue).

Una volta creata l’interfaccia, implementiamo il metodo *ls* richiamando il metodo *print* sull’oggetto *FileSystemPrinter*. Il codice non compilerà perché il metodo non esiste ancora. Sempre usando il Quickfix lo facciamo creare a Eclipse dentro l’interfaccia (date un nome sensato al parametro del metodo *print*, ad es., *message*):

```
public void ls(FileSystemPrinter printer) {
    printer.print("File: " + name);
}

}
```

A screenshot of the Eclipse IDE showing the `ls` method implementation. A red squiggly line under the `printer.print` call triggers a Quickfix menu. The menu has one option: "Create method 'print(String)' in type 'FileSystemPrinter'" (highlighted in blue).

Il metodo creato nell'interfaccia dovrà essere modificato in modo che il nome del parametro di tipo stringa abbia un nome sensato, ad es., "message".

Alla fine avremo:

```
package mp.exercise.filesystem;

public interface FileSystemPrinter {

    void print(String message);

}

public class FileSystemFile {

    private String name;

    public FileSystemFile(String name) {
        this.name = name;
    }

    public void ls(FileSystemPrinter printer) {
        printer.print("File: " + name);
    }

}
```

Scriviamo il test per il metodo *ls*.

Al momento non abbiamo nessuna implementazione di *FileSystemPrinter*. Però possiamo comunque testare il metodo *ls*: basta creare una classe che implementa l'interfaccia (un'implementazione fittizia detta "mock"), nella directory sorgente dei test (in quanto si tratta di un'implementazione fittizia che serve solo ai test), col minimo indispensabile per permetterci di testare agevolmente la correttezza di *ls*. Ad es., ci basta un'implementazione che memorizza tutte le stringhe passate a *print*, magari separate da un carattere "a capo", "\n", e un metodo per recuperare tutto quello che è stato memorizzato. Alla fine del test, ci basta verificare che le stringhe passate abbiano la forma che ci aspettiamo.

Ad es.,

```
public class MockFileSystemPrinter implements FileSystemPrinter {

    private StringBuilder builder = new StringBuilder();

    @Override
    public void print(String message) {
        builder.append(message + "\n");
    }

    @Override
    public String toString() {
        return builder.toString();
    }

}
```

ATTENZIONE: questa implementazione "mock" serve solo nei test, non fa parte dei sorgenti principali dell'applicazione. Mettere questa implementazione nella directory sorgente "src" sarebbe un grave errore!

Nella classe di test dobbiamo inizializzare tale istanza in un metodo *@Before*, così ogni metodo di test userà sempre un oggetto nuovo e i vari metodi di test saranno indipendenti l'uno dall'altro (ricordate che ogni test deve essere eseguibile in isolamento e non subire “side effect” da parte di altri test). E possiamo testare la nostra classe *FileSystemFile*:

```
package mp.exercise.filesystem;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class FileSystemFileTest {

    private MockFileSystemPrinter printer;

    @Before
    public void setup() {
        printer = new MockFileSystemPrinter();
    }

    @Test
    public void testFileLs() {
        FileSystemFile file = new FileSystemFile("aFile.txt");
        file.ls(printer);
        assertEquals("File: aFile.txt\n", printer.toString());
    }
}
```

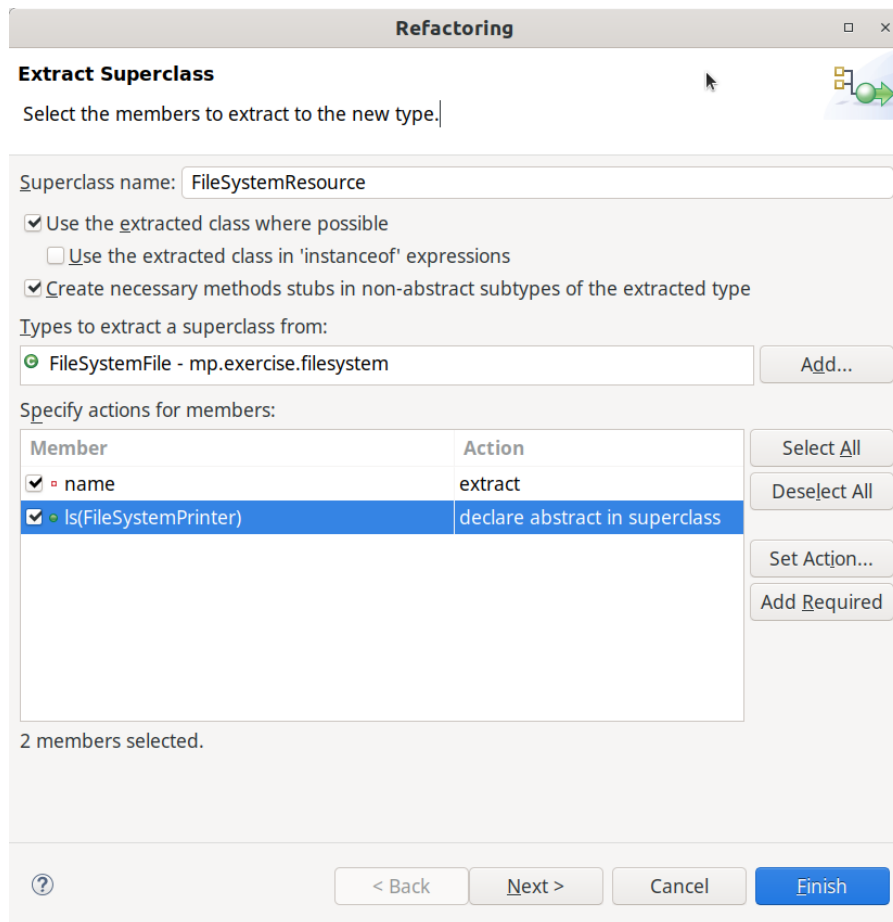
NOTA: Ad *ls* passiamo un *FileSystemPrinter* invece di assumere che il risultato venga sempre stampato su *System.out*. In questo modo il metodo sarà in grado di stampare su un qualsiasi oggetto la cui classe implementi questa interfaccia. Oltre a rendere il nostro codice metodo più generico e rendere la classe aperta a estensioni, rende il metodo *ls* molto più facilmente testabile: come abbiamo visto, non importa nemmeno avere già un'implementazione di questa interfaccia, perché ne possiamo creare una fittizia da usare facilmente nei test. Se invece *ls* non prendesse un parametro e usasse all'interno sempre *System.out* allora testarlo sarebbe un po' più complesso (nonostante fattibile: in Java si può “catturare” durante l'esecuzione l'output di *System.out*, in particolare lo si può riassegnare dinamicamente; ma questo, appunto, è molto più complesso). Esistono dei framework appositi di test per creare molto facilmente e “al volo” delle implementazioni “mock”. Non vedremo in questo corso tali framework. Inoltre, se *ls* usasse sempre *System.out*, la sua implementazione NON sarebbe estendibile senza modifiche.

2 Estrarre classe base astratta

Vogliamo una classe base astratta sia per le foglie (file) che per gli oggetti composti (directory). La vogliamo creare partendo da *FileSystemFile*. Tale classe base, che chiameremo *FileSystemResource*, contiene il campo comune *name*, e un *getter* al momento *protected* e la dichiarazione (astratta) del metodo *ls* visto prima. Usare le tecniche viste alla scorsa esercitazione per effettuare tale refactoring. I test non devono richiedere modifiche e devono continuare a passare.

In modo simile a come visto alla scorsa esercitazione, eseguiamo i seguenti passi.

Estrarre da *FileSystemFile* la superclasse astratta *FileSystemResource* col campo *name*. Anche il metodo *ls* deve essere estratto, ma come “Action” si deve specificare che nella superclasse deve essere *abstract*:



Si otterrà (ricordate che il costruttore di una classe astratta dovrebbe essere protetto):

```
package mp.exercise.filesystem;

public abstract class FileSystemResource {
    protected String name;

    public abstract void ls(FileSystemPrinter printer);

    protected FileSystemResource() {
        super();
    }
}
```

E la classe *FileSystemFile* sarà modificata come:

```
public class FileSystemFile extends FileSystemResource {
    public FileSystemFile(String name) {
        this.name = name;
    }

    @Override
    public void ls(FileSystemPrinter printer) {
        printer.print("File: " + name);
    }
}
```

```
}  
}
```

Rilanciare il test che avrà successo. Notare che siccome abbiamo selezionato la spunta “Use extracted class where possible”, il test è stato modificato usando *FileSystemResource* come tipo della variabile usata nel test. Il test è stato modificato come conseguenza del refactoring, non abbiamo dovuto modificarlo manualmente.

Ora dobbiamo sistemare il campo `protected` in modo simile a come avevamo fatto alla scorsa esercitazione. Prima di tutto, in *FileSystemResource* aggiungiamo (tramite il tool di Eclipse) il costruttore “using fields”, e rimuoviamo quello senza argomenti:

```
public abstract class FileSystemResource {  
    protected String name;  
  
    protected FileSystemResource(String name) {  
        this.name = name;  
    }  
  
    public abstract void ls(FileSystemPrinter printer);  
}
```

Aggiustiamo *FileSystemFile* in modo da passare il parametro al costruttore della superclasse:

```
public class FileSystemFile extends FileSystemResource {  
    public FileSystemFile(String name) {  
        super(name);  
    }  
    ...  
}
```

Rilanciare i test.

Usando il tool “Encapsulate Field” (vedere la scorsa esercitazione), aggiungiamo il *getter* `protected` nella superclasse, e rendiamo privato il campo `name`:

```
public abstract class FileSystemResource {  
    private String name;  
  
    protected FileSystemResource(String name) {  
        this.name = name;  
    }  
  
    protected String getName() {  
        return name;  
    }  
  
    public abstract void ls(FileSystemPrinter printer);  
}  
  
public class FileSystemFile extends FileSystemResource {  
    public FileSystemFile(String name) {
```

```

        super(name);
    }

    @Override
    public void ls(FileSystemPrinter printer) {
        printer.print("File: " + getName());
    }
}

```

Rilanciare i test.

3 Directory

Implementiamo adesso il componente *Composite*: la *directory*. La classe *FileSystemDirectory*, eredita da *FileSystemResource*, quindi dovrà avere un costruttore appropriato. Dovrà avere anche un campo per memorizzare i “contenuti”, i “figli”, ad es., *contents*. Al momento non abbiamo richieste particolari sulla struttura della collezione quindi usiamo *Collection<...>*. Che tipo mettiamo al posto di “...”? Ricordate che stiamo implementando il pattern *Composite*. Al momento dobbiamo implementare solo il metodo *ls*: ancora non implementiamo i metodi per gestire i “figli”, basandosi sul pattern *Composite*. Ricordate inoltre che nel pattern il composto deve delegare l’operazione ai figli, eventualmente facendo qualcosa prima e/o dopo. Per distinguere il fatto che stiamo “stampando” una directory, la prima stringa passata al metodo *print* dovrebbe essere “Directory: <nome>”. Poi, come appena detto, l’operazione dovrà essere delegata ai figli. Siccome implementeremo al momento solo *ls*, e non le operazioni sui figli (aggiunta, rimozione, ecc.), come potremmo fare per testare *ls*?

In particolare, dobbiamo testare diverse situazioni:

1. caso in cui la directory è vuota
2. caso in cui contiene almeno un file
3. caso in cui contiene sottodirectory

Come possiamo testare la classe se non possiamo usare le operazioni per la gestione dei figli?

Implementiamo la classe come derivata di *FileSystemResource*, usando il wizard di Eclipse con le apposite opzioni (vedere esercitazione della volta scorsa):

```

public class FileSystemDirectory extends FileSystemResource {

    public FileSystemDirectory(String name) {
        super(name);
        // TODO Auto-generated constructor stub
    }

    @Override
    public void ls(FileSystemPrinter printer) {
        // TODO Auto-generated method stub
    }

}

```

Per il momento dichiariamo il campo *contents* come *Collection<FileSystemResource>* (infatti una directory può contenere sia file che sottodirectory, quindi, come previsto nel pattern *Composite*,

usiamo il tipo base *FileSystemResource*). Non avendo richieste particolari sulla struttura della collezione è opportuno usare *Collection* e non *List*. Per il momento inizializziamo direttamente il campo con un *ArrayList*. Più tardi (in altre lezioni/esercitazioni) ci preoccuperemo di rimuovere questa dipendenza forte verso un tipo concreto. Implementiamo in modo opportuno anche *ls*, in modo che, dopo aver “stampato” il nome della directory, inoltri *ls* a tutti i contenuti:

```
public class FileSystemDirectory extends FileSystemResource {  
    private Collection<FileSystemResource> contents = new ArrayList<>();  
  
    public FileSystemDirectory(String name) {  
        super(name);  
    }  
  
    @Override  
    public void ls(FileSystemPrinter printer) {  
        printer.print("Directory: " + getName());  
        contents.forEach(resource -> resource.ls(printer));  
    }  
}
```

Anche nei prossimi test utilizzeremo la nostra implementazione “mock” per il printer: *MockFileSystemPrinter*.

Dobbiamo testare diverse situazioni:

1. caso in cui la directory è vuota
2. caso in cui contiene almeno un file
3. caso in cui contiene sottodirectory

Il primo test è semplice, essendo la directory vuota:

```
package mp.exercise.filesystem;  
...  
public class FileSystemDirectoryTest {  
    private MockFileSystemPrinter printer;  
  
    @Before  
    public void setup() {  
        printer = new MockFileSystemPrinter();  
    }  
  
    @Test  
    public void testLsEmptyDirectory() {  
        FileSystemDirectory dir = new FileSystemDirectory("aDir");  
        dir.ls(printer);  
        assertEquals("Directory: aDir\n", printer.toString());  
    }  
}
```

Non avendo le operazioni per gestire i figli, come possiamo inizializzare i contenuti di un *FileSystemDirectory*? Possiamo aggiungere un *getter* per *contents* con visibilità solo a livello di

package. In questo modo non lo esponiamo al “mondo”, e mettendo i test nello stesso package (anche se in directory sorgente differenti), possiamo accedere al *getter* dai test.

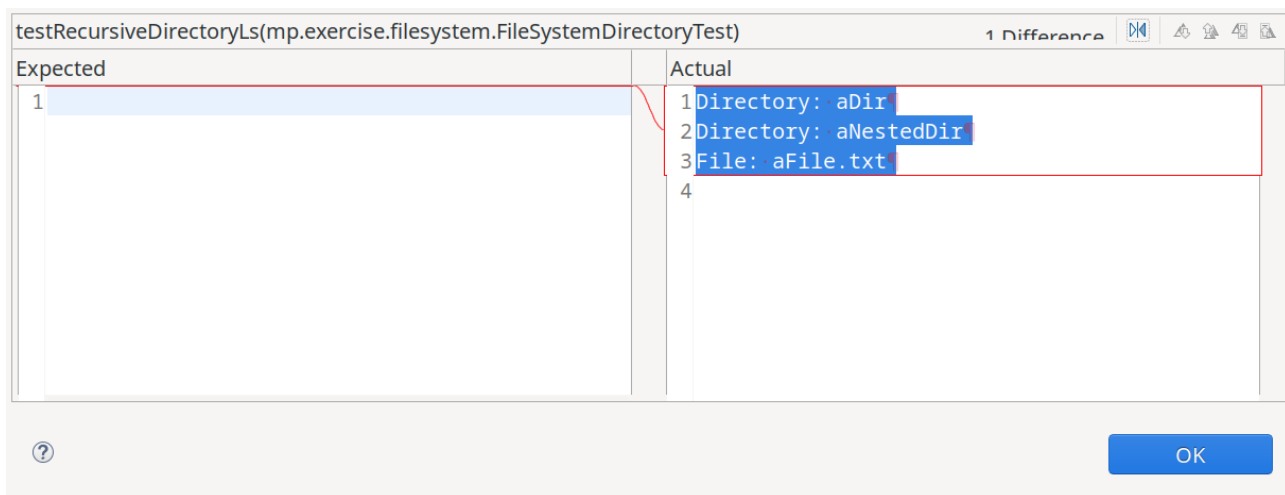
```
public class FileSystemDirectory extends FileSystemResource {  
    private Collection<FileSystemResource> contents = new ArrayList<>();  
    ...  
    /**  
     * Just for testing, package-private  
     */  
    Collection<FileSystemResource> getContents() {  
        return contents;  
    } ...  
}
```

Avendo un riferimento alla collection interna la possiamo inizializzare per i nostri due rimanenti test:

```
@Test  
public void testNonEmptyDirectoryLs() {  
    FileSystemDirectory dir = new FileSystemDirectory("aDir");  
    dir.getContents().add(new FileSystemFile("aFile.txt"));  
    dir.ls(printer);  
    assertEquals(  
        "Directory: aDir\n"+  
        "File: aFile.txt\n",  
        printer.toString());  
}  
  
@Test  
public void testRecursiveDirectoryLs() {  
    FileSystemDirectory dir = new FileSystemDirectory("aDir");  
    FileSystemDirectory nestedDir = new FileSystemDirectory("aNestedDir");  
    dir.getContents().add(nestedDir);  
    nestedDir.getContents().add(new FileSystemFile("aFile.txt"));  
    dir.ls(printer);  
    assertEquals(  
        "Directory: aDir\n"+  
        "Directory: aNestedDir\n"+  
        "File: aFile.txt\n",  
        printer.toString());  
}
```

Volendo esistono anche altre tecniche di testing che evitano l’uso di un *getter* package-private, ma non le vedremo in questo corso.

SUGGERIMENTO: soprattutto quando nei test si usa *assertEquals* con stringhe, è laborioso specificare la stringa che ci aspettiamo ed è anche difficile specificarla in modo corretto. Si può in prima istanza specificare una stringa vuota, lanciare il test che fallirà, fare doppio click sulla “Failure Trace” in corrispondenza di “ComparisonFailure”. Eclipse aprirà una finestra di dialogo con la stringa attesa (in questo caso abbiamo specificato la stringa vuota) e quella effettiva:



A questo punto basta selezionare tutta la stringa “Actual” (incluso l’ultimo carattere di invio), copiarla, e incollarla nel test dentro le “”. Eclipse penserà a trasformare la stringa con la concatenazione e i caratteri “\n”. Otterremo così il risultato mostrato sopra. OVVIAMENTE, prima di fare ciò ci dobbiamo assicurare che la stringa “Actual” sia effettivamente corretta.

4 Gestione contenuti: add

Implementare nella classe *FileSystemDirectory* il metodo *void add* per aggiungere contenuti nel *Composite*. Quindi che tipo deve avere il parametro? Per testarne la correttezza, NON usiamo *ls* (quindi come possiamo testare *add*?).

L’implementazione è semplicemente:

```
public void add(FileSystemResource resource) {
    contents.add(resource);
}
```

Per testarlo ci affidiamo al *getter*: dopo aver aggiunto una risorsa con *add* controlliamo che sia nella collezione recuperata tramite il *getter*:

```
@Test
public void testDirectoryAdd() {
    FileSystemDirectory dir = new FileSystemDirectory("aDir");

    FileSystemResource res = new FileSystemFile("a file");
    dir.add(res);

    Collection<FileSystemResource> contents = dir.getContents();
    assertEquals(1, contents.size());
    assertTrue(contents.contains(res));
}
```

Notare che ci assicuriamo che il file aggiunto nel test è l’unico elemento nella collezione (per questo asseriamo anche che la collezione abbia dimensione 1).

IMPORTANTE: In generale NON si dovrebbe MAI testare un metodo di una classe usando un altro metodo della stessa classe (le uniche eccezioni sono ovviamente i *getter* e i *setter* che possono essere utilizzati durante i test, se presenti nella classe). Quindi, per testare *add* non si dovrebbe effettuare la verifica usando *ls*. Se *ls* contenesse un bug potrebbe succedere che il test per *add*

fallisca nonostante *add* sia corretto (falso negativo) oppure che il test per *add* abbia successo nonostante *add* contenga un bug (falso positivo). Inoltre, potremmo implementare *add* prima di *ls* e quindi non potremmo testare *add* se usassimo *ls* per fare le verifiche.

OPZIONALE: Usare AssertJ per scrivere le asserzioni sulle collezioni in modo più “fluido” e leggibile.

5 Remove

Similmente, implementare anche il metodo *void remove*, col parametro appropriato. Nei test per *remove*, NON dobbiamo usare *add*.

Anche l’implementazione di *remove* è banale: si delega direttamente alla *Collection*:

```
public void remove(FileSystemResource resource) {
    contents.remove(resource);
}
```

Possiamo ignorare il valore di ritorno di *Collection.remove* in questa esercitazione.

Per testarla, nuovamente, ci affidiamo al *getter* sia per inizializzare manualmente la collezione di contenuti sia per verificare l’effettiva rimozione. È fondamentale, NON usare *add* per inizializzare i contenuti, per quanto detto sopra. Per rendere il test più attendibile, inseriamo due elementi e ne rimuoviamo uno, e poi verifichiamo che sia stato rimosso solo quello che ci interessa:

```
@Test
public void testDirectoryRemove() {
    FileSystemDirectory dir = new FileSystemDirectory("aDir");

    FileSystemResource notToRemove = new FileSystemFile("file1");
    FileSystemResource toRemove = new FileSystemFile("file2");

    Collection<FileSystemResource> contents = dir.getContents();
    contents.addAll(Arrays.asList(notToRemove, toRemove));

    dir.remove(toRemove);
    assertEquals(1, contents.size());
    assertTrue(contents.contains(notToRemove));
}
```

6 Ricerca per nome

In questo esempio non ha molto senso implementare l’accesso a un elemento tramite indice (come invece viene fatto a volte col pattern *Composite*). È meglio implementare una ricerca per nome della risorsa:

Optional<FileSystemResource> findByName(String name)

Questo va implementato solo nella classe del *Composite*, quindi in *FileSystemDirectory*. Per il momento, NON implementeremo una ricerca ricorsiva nelle sottodirectory, ma solo sul contenuto della directory su cui invochiamo il metodo (ci occuperemo della ricerca ricorsiva in un’eventuale altra esercitazione). La ricerca non distingue fra file e directory ovviamente. È importante testarla sia quando ci si aspetta di trovare un file sia quando ci si aspetta di non trovarlo.

Per implementare il metodo usiamo stream e lambda:

```
public Optional<FileSystemResource> findByName(String name) {  
    return contents.stream()  
        .filter(resource -> resource.getName().equals(name))  
        .findFirst();  
}
```

E come sempre testiamo il metodo usando il *getter* per accedere ai contenuti. Testiamo sia quando ci si aspetta di trovare un file sia quando ci si aspetta di non trovarlo (per semplicità possiamo testare entrambe le situazioni in un unico test; in generale però si dovrebbe avere un metodo test separato per ogni situazione):

```
@Test  
public void testDirectoryFindByName() {  
    FileSystemDirectory dir = new FileSystemDirectory("a dir");  
  
    Collection<FileSystemResource> contents = dir.getContents();  
    FileSystemResource res1 = new FileSystemFile("file1");  
    FileSystemResource res2 = new FileSystemFile("file2");  
    contents.addAll(Arrays.asList(res1, res2));  
  
    assertEquals(res2, dir.findByName("file2").get());  
    assertFalse(dir.findByName("file3").isPresent());  
}
```

Notare che è bene scrivere un test che forzi il predicato passato a *filter* a essere valutato sia a true che a false. Per questo, è bene che l'elemento da trovare NON sia il primo nella lista dei contenuti.

IMPORTANTE: nei test è lecito usare “scorciatoie” come accedere direttamente all'elemento di un *Optional* con *get*. Nel codice vero e proprio, invece, si deve usare l'API giusta: ricordate che NON si dovrebbe accedere direttamente al contenuto di un *Optional* con *get* (in quanto se l'elemento non è presente si ottiene un'eccezione – nei test, appunto, questo non è un grosso problema in quanto in tal caso il test fallirà), ma si dovrebbe usare *ifPresent*, *orElse*, ecc., a seconda del contesto.

7 Metodi di Object

Implementare *toString*, *equals* e *hashCode*, secondo le norme viste a “Programmazione” nelle varie classi. Potete usare i meccanismi di Eclipse per farveli generare automaticamente. Attenzione però a come Eclipse vi genera questi metodi: in alcuni casi non tiene conto dell'ereditarietà, quindi dovrete aggiustarli in modo opportuno (dobbiamo implementare *toString*, *equals*, e *hashCode* proprio in tutte le classi o possiamo riusare qualcosa in qualche classe?). Nel caso della directory si può generare anche la stringa dei contenuti in *toString*. Non è strettamente necessario testare questi metodi se li generate automaticamente. Comunque, a livello didattico, conviene scrivere qualche test per questi metodi.

Vedere il codice dell'esercizio completo su Moodle.

8 File copy

Nella classe *FileSystemFile* implementiamo il metodo *createCopy* con tipo di ritorno *FileSystemResource* che crea un nuovo oggetto *FileSystemFile* con lo stesso nome del file su cui viene invocato. Nei test ci dobbiamo assicurare che l'oggetto restituito

1. NON sia lo stesso oggetto di quello originale e
2. sia “uguale” (tramite *equals*, implementato al passo precedente).

Implementiamo il metodo semplicemente costruendo un nuovo oggetto usando il nome di quello corrente:

```
public FileSystemResource createCopy() {  
    return new FileSystemFile(getName());  
}
```

Nei test ci dobbiamo assicurare che l'oggetto restituito NON sia lo stesso oggetto di quello originale, ma che siano *equals*:

```
@Test  
public void testFileCreateCopy() {  
    FileSystemFile original = new FileSystemFile("aFile");  
    FileSystemResource copy = original.createCopy();  
  
    assertNotSame(original, copy);  
    assertEquals(original, copy);  
}
```

9 Directory copy

Dopo aver portato la dichiarazione astratta di *createCopy* nella superclasse *FileSystemResource*, implementare la copia ricorsiva anche in *FileSystemDirectory*. È poi possibile, nel metodo *createCopy*, avere un tipo di ritorno più specializzato in *FileSystemFile* e *FileSystemDirectory* in modo da sapere staticamente che la copia di un file è un file e la copia di una directory è una directory (invece di una generica risorsa)?

Attenzione: nei test si deve controllare che due oggetti copia siano uguali secondo *equals* ma che NON siano lo stesso oggetto in memoria. Nel caso della copia ricorsiva di una directory, testare che non si tratti dello stesso oggetto richiede di scrivere un po' di codice nei test perché il controllo deve essere fatto ricorsivamente sui contenuti delle due copie di directory.

Usiamo “Pull Up” per portare la dichiarazione astratta di *createCopy* in *FileSystemResource* (come fatto in precedenza in questa esercitazione, dobbiamo selezionare “declare abstract in superclass”). Questo refactoring inserirà anche un'implementazione vuota di *createCopy* in *FileSystemDirectory*.

Implementiamo la copia ricorsivamente (sfruttando *createCopy*) usando gli stream con *map* e *collect*:

```
@Override  
public FileSystemDirectory createCopy() {  
    FileSystemDirectory copy = new FileSystemDirectory(getName());  
    copy.getContents().addAll(  
        contents.stream()
```

```

        .map(FileSystemResource::createCopy)
        .collect(Collectors.toList());
    }
    return copy;
}

```

Notare che sfruttando la covarianza di Java durante la ridefinizione di metodo specifichiamo come tipo di ritorno *FileSytemDirectory*.

Il test corrispondente usa *assertEquals* che si basa su *equals* di *FileSystemDirectory* (che deve essere già stato implementato in modo appropriato, confrontando anche i figli ricorsivamente; vedi uno dei punti precedenti dell'esercitazione); inoltre ispezioniamo ricorsivamente la struttura delle due directory per assicurarsi che siano state effettivamente effettuate delle copie, cioè che non si tratti dello stesso oggetto in memoria (usiamo un metodo privato per effettuare queste verifiche, in modo da migliorare la leggibilità dei test e perché può tornare utile per test futuri):

```

@Test
public void testDirectoryCopy() {
    FileSystemDirectory original = new FileSystemDirectory("a dir");
    FileSystemDirectory nested = new FileSystemDirectory("a nested dir");
    nested.getContents().add(new FileSystemFile("second file"));
    original.getContents()
        .addAll(Arrays.asList(new FileSystemFile("first file"), nested));

    FileSystemDirectory copy = original.createCopy();

    assertEquals(original, copy);
    assertEffectiveCopyDirectory(original, copy);
}

private void assertEffectiveCopyDirectory(FileSystemDirectory dir1,
                                          FileSystemDirectory dir2) {
    assertNotSame(dir1, dir2);
    Iterator<FileSystemResource> contents1 = dir1.getContents().iterator();
    Iterator<FileSystemResource> contents2 = dir2.getContents().iterator();
    // si assume che le due collezioni abbiano la stessa lunghezza,
    // e che a un elemento della prima corrisponda un elemento della
    // seconda dello stesso tipo.
    // Questo e' vero se e' gia' stato usato assertEquals
    while (contents1.hasNext()) {
        FileSystemResource res1 = contents1.next();
        FileSystemResource res2 = contents2.next();
        if (res1 instanceof FileSystemDirectory) {
            assertEffectiveCopyDirectory
                ((FileSystemDirectory) res1, (FileSystemDirectory) res2);
        } else {
            assertNotSame(res1, res2);
        }
    }
}

```

IMPORTANTE: Come detto in un punto precedente, nel test facciamo uso di cast “brutali” senza un *instanceof*. Nei test questo è ammesso ed è giusto che il test fallisca con una *ClassCastException* se il cast non può essere fatto come invece ci si aspettava. Al contrario, nel codice vero e proprio non si dovrebbero usare down-cast e nemmeno *instanceof*. Si dovrebbero usare pattern appropriati (come avremo modo di vedere durante il corso).