Esercizio file system e Visitor

In questo esercizio useremo nuovamente il pattern *Visitor* nella versione metodi void. Ricordate che soprattutto per i visitor con metodi void è normale mantenersi uno stato. In tal caso, un oggetto visitor deve essere utilizzato una e una sola volta. Partiamo da quello che avevamo già implementato nell'esempio del file system (esercitazione su pattern *Composite*). Questa esercitazione è pensata anche per farvi prendere pratica con AssertJ. Inoltre, non dovremo mai usare ne' *instanceof* ne' cast.

Le nuove classi che implementeremo dal punto 3 in poi dovranno stare in un package differente *utils* (quindi, secondo i nomi dell'altra esercitazione nel package *mp.exercise.filesystem.utils*). La stessa cosa vale per i test corrispondenti. In questo modo, non potranno accedere a eventuali metodi package-private del package principale *mp.exercise.filesystem*.

1 Directory iterator

Nella classe *FileSystemDirectory* creare il metodo *iterator*() che restituisce un Java *Iterator* di *FileSystemResource*. Quindi su una directory, questo iteratore permette di iterare su tutti i contenuti della directory, file ed eventuali sotto-directory, senza però andare in profondità nelle sotto-directory. L'implementazione del metodo sarà molto semplice, ma assicuratevi di testarla con AssertJ.

IMPORTANTE: Ricordate che ai fini del progetto, l'uso dell'iteratore di Java NON viene considerato come l'applicazione del pattern *Iterator.*

L'implementazione è semplicemente:

```
    public Iterator<FileSystemResource> iterator() {
        return contents.iterator();
    }
```

Il test è

```
@Test
public void testRecursiveDirectoryIterator() {
    FileSystemDirectory dir = new FileSystemDirectory("aDir");
    FileSystemDirectory nestedDir = new FileSystemDirectory("aNestedDir");
    FileSystemFile file1 = new FileSystemFile("aFile.txt");
    dir.getContents().add(file1);
    dir.getContents().add(nestedDir);
    nestedDir.getContents().add(new FileSystemFile("aNotherFile.txt"));
    assertThat(dir.iterator())
        .toIterable()
        .containsExactlyInAnyOrder(file1, nestedDir);
}
```

Notare che il test si assicura che l'iteratore non vada "in profondità", infatti l'iteratore permette di accedere alla directory annidata, ma non ai suoi contenuti.

Implementare un *iterator* che iteri ricorsivamente non è banale e per gli scopi di questo corso non è nemmeno interessante. Potete provare a implementarlo per esercizio, magari una volta che avete implementato il *visitor* nei prossimi passi. Comunque, l'iterator restituisce anche eventuali sottodirectory, quindi il *visitor* potrà a sua scelta andare in profondità se necessario, come vedremo nei prossimi passi.

2 Visitor

Implementare l'interfaccia *FileSystemVisitor* coi metodi void *visitFile* e *visitDirectory* nello stesso package di *FileSystemResource*, *FileSystemFile*, ecc. Aggiungere i vari *accept* (dichiarando il metodo come astratto in *FileSystemResource*).

Vedere l'implementazione nei sorgenti della soluzione.

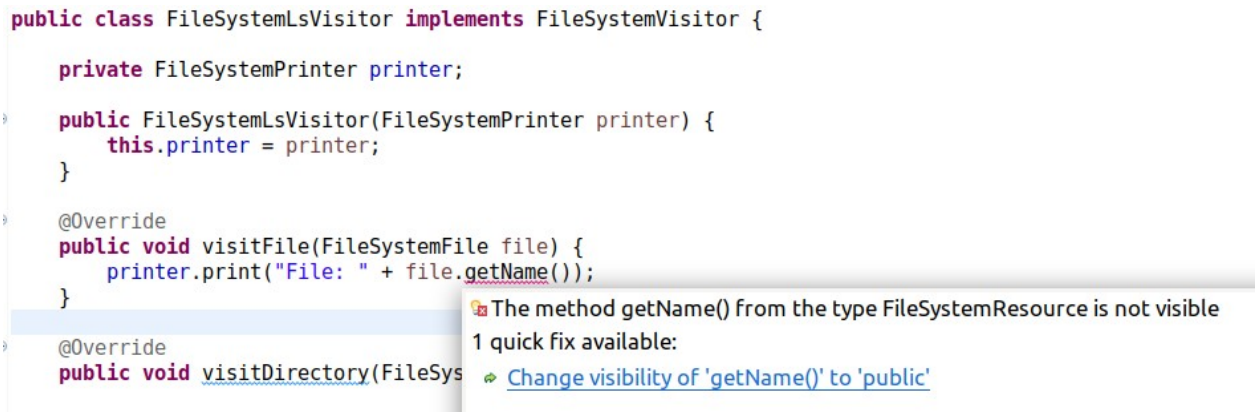
3 Ls tramite visitor

Implementare la classe *FileSystemLsVisitor*. Tale visitor deve implementare lo stesso comportamento di *ls* nell'esercitazione precedente. Deve quindi usare *FileSystemPrinter*, che deve essere passato al costruttore. In particolare, nel caso delle directory, deve andare in profondità listando tutti i contenuti (anche delle sottodirectory). Il metodo *ls* deve essere rimosso dalla gerarchia di *FileSystemResource* e i test esistenti devono quindi essere adattati: le funzionalità del vecchio *ls* ora devono essere testate in *FileSystemLsVisitorTest*, che deve essere nel package *mp.exercise.filesystem.utils* (ovviamente nella directory sorgente dei test). Il "mock" per *FileSystemPrinter* che avevamo implementato alla precedente esercitazione deve essere riusato in questi nuovi test. La classe astratta *FileSystemResource* dovrà essere adattata per permettere al visitor di accedere al nome della risorsa (ricordate che il visitor "rompe l'incapsulamento"). Una volta che tutto è testato, tramite refactoring, spostate *FileSystemPrinter* nel package *utils*. Allo stesso modo il "mock" deve essere spostato nel package *utils* nella directory dei test.

Il visitor sarà di questa forma:

```
public class FileSystemLsVisitor implements FileSystemVisitor {

    private FileSystemPrinter printer;

    public FileSystemLsVisitor(FileSystemPrinter printer) {
        this.printer = printer;
    }

    @Override
    public void visitFile(FileSystemFile file) {
        printer.print("File: " + file.getName());
    }

    @Override
    public void visitDirectory(FileSystemDirectory dir) {
        printer.print("Directory: " + dir.getName());
        Iterator<FileSystemResource> iterator = dir.iterator();
        while (iterator.hasNext())
            iterator.next().accept(this);
    }

}
```

Quando si implementano i metodi *visit* in *FileSystemLsVisitor* si proverà ad accedere a *getName()* ottenendo un errore perché quel *getter* è protetto in *FileSystemResource*. Usare il quickfix per renderlo pubblico.



Rimuovere *ls* dalla gerarchia di *FileSystemResource* e adattare i test: siccome *ls* non fa più parte di *FileSystemResource* i test corrispondenti non compilano più. Le medesime situazioni vanno adesso testate in *FileSystemLsVisitorTest*. In pratica si tratta dei test precedenti per *ls* adattati a questa nuova classe, usando ancora *MockFileSystemPrinter*:

```
public class FileSystemLsVisitorTest {

    private MockFileSystemPrinter printer;
    private FileSystemLsVisitor visitor;

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

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

    @Test
    public void testEmptyDirectoryLs() {
        FileSystemDirectory dir = new FileSystemDirectory("aDir");
        dir.accept(visitor);
        assertEquals("Directory: aDir\n", printer.toString());
    }

    @Test
    public void testNonEmptyDirectoryLs() {
        FileSystemDirectory dir = new FileSystemDirectory("aDir");
        dir.add(new FileSystemFile("aFile.txt"));
        dir.accept(visitor);
        assertEquals(
                "Directory: aDir\n"+
                "File: aFile.txt\n",
            printer.toString());
    }

    @Test
    public void testRecursiveDirectoryLs() {
```

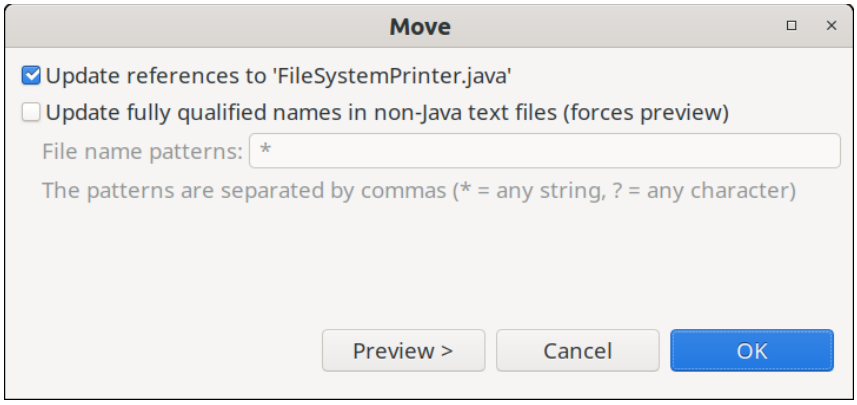
```
        FileSystemDirectory dir = new FileSystemDirectory("aDir");
        FileSystemDirectory nestedDir = new FileSystemDirectory("aNestedDir");
        dir.add(nestedDir);
        nestedDir.add(new FileSystemFile("aFile.txt"));
        dir.accept(visitor);
        assertEquals(
                "Directory: aDir\n"+
                "Directory: aNestedDir\n"+
                "File: aFile.txt\n",
            printer.toString());
    }
```

L'istanza di *FileSystemLsVisitor* viene sempre creata nel metodo annotato con *@Before* e gli viene passato il "mock" printer.

È da notare che adesso questa classe testa *FileSystemLsVisitor*, quindi è lecito chiamare direttamente *dir.add* per inizializzare l'oggetto directory. Invece, a suo tempo, per testare *ls* dentro *FileSystemDirectory* non dovevamo chiamare un metodo della stessa classe (*add*) e usavamo il *getter* package-private. Tale *getter* in questo test non lo possiamo nemmeno chiamare perché siamo in un altro package, ed è giusto così.

Infatti, quando si testa una classe *A* che fa uso di un altro tipo (eventualmente concreto) *B*, ci si deve concentrare solo sulla verifica del comportamento di *A* assumendo che *B* si comporti correttamente (cioè che sia già stato testato, nel caso sia un tipo concreto, o che lo sarà quando sarà implementato).

Infine per spostare le classi nel package utils si può usare il refactoring "Move", oppure semplicemente fare drag-and-drop della classe nel nuovo package: i riferimenti saranno automaticamente aggiornati, selezionando l'opzione apposita nella finestra che appare, ad es., per *FileSystemPrinter*:



Fare la stessa procedura per *MockFileSystemPrinter.*

4 Ls tramite visitor ma non ricorsivo

Implementare la classe *FileSystemNonRecursiveLsVisitor* che si comporta in modo simile a *FileSystemLsVisitor*: nel caso di un file "stampa" il nome del file, nel caso di una directory si comporta come il precedente ma poi non entra, ricorsivamente, dentro le eventuali sotto-directory (e quindi i nomi di eventuali sotto-directory vengono comunque stampati, ma non i loro contenuti). Ricordate: non dovremo mai usare ne' *instanceof* ne' cast. Suggerimento: un visitor, soprattutto se

ha i metodi void, può avere un suo stato. In tal caso, un oggetto visitor deve essere utilizzato una e una sola volta.

```
public class FileSystemNonRecursiveLsVisitor implements FileSystemVisitor {

    private FileSystemPrinter printer;
    private boolean alreadyVisitedDirectory = false;

    public FileSystemNonRecursiveLsVisitor(FileSystemPrinter printer) {
        this.printer = printer;
    }

    @Override
    public void visitFile(FileSystemFile file) {
        printer.print("File: " + file.getName());
    }

    @Override
    public void visitDirectory(FileSystemDirectory dir) {
        printer.print("Directory: " + dir.getName());
        if (alreadyVisitedDirectory)
            return;
        alreadyVisitedDirectory = true;
        Iterator<FileSystemResource> iterator = dir.iterator();
        while (iterator.hasNext())
            iterator.next().accept(this);
    }

}
```

Notare che siccome non vogliamo usare instanceof, dobbiamo mantenerci l'informazione riguardo al fatto che abbiamo già visitato una directory. Quando visitiamo una directory, se tale valore è false, allora dobbiamo visitare ricorsivamente i suoi contenuti (dopo aver messo a true tale informazione), in quanto è la prima volta che visitiamo una directory (che è quella principale). Se il valore è true invece non dobbiamo visitare ricorsivamente i suoi contenuti: vuol dire che stiamo visitando una directory che fa parte della directory principale che abbiamo già visitato.

Per i test conviene copiare quelli precedenti e adattare quelli che devono essere adattati. Ad es., questo test deve essere adattato perché il contenuto della directory annidata non viene stampato:

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

ATTENZIONE: implementare *FileSystemNonRecursiveLsVisitor* come sottoclasse di *FileSystemLsVisitor* o viceversa viola LSP.

Se si volesse riusare il codice nelle due classi? Usare una classe base astratta parametrizzata sulla strategy per continuare nella ricorsione. Oppure, si potrebbe usare una classe astratta in cui definiamo un *template method* basato su operazioni che vengono ridefinite nelle sottoclassi.

NOTA: Come abbiamo detto sopra, per i test conviene copiare quelli precedenti e modificare quelli che devono essere adattati. Avremo quindi due classi di test molto simili (in questo esempio differiranno solo per un test), con molto codice duplicato. Questo NON è un problema: nei test, la duplicazione di codice non solo è ammissibile, ma è anche consigliata se rende i test più semplici da leggere e mantenere. Anzi, non si dovrebbero introdurre astrazioni nei test allo scopo di ridurre il codice duplicato, in quanto renderebbe i test più difficili da gestire e mantenere.

5 Recuperare solo i file di una directory

Implementare la classe *DirectoryFilesCollectionSupplier*, che prende un *FileSystemDirectory* come parametro del costruttore e implementa *java.util.function.Supplier<Collection<FileSystemFile>>*. Deve quindi implementare il metodo astratto di quell'interfaccia in modo da restituire una collezione dei soli file (ma non di eventuali sottodirectory, ne' di file contenuti in sottodirectory) della directory passata al costruttore. Notare che questa classe NON implementa *FileSystemVisitor* (ma ne implementerà e userà al suo interno).

```
public class DirectoryFilesCollectionSupplier
             implements Supplier<Collection<FileSystemFile>> {

    private FileSystemDirectory dir;

    public DirectoryFilesCollectionSupplier(FileSystemDirectory dir) {
        this.dir = dir;
    }

    @Override
    public Collection<FileSystemFile> get() {
        Iterator<FileSystemResource> iterator = dir.iterator();
        class FileFilter implements FileSystemVisitor {
            ArrayList<FileSystemFile> files = new ArrayList<>();

            @Override
            public void visitFile(FileSystemFile file) {
                files.add(file);
            }

            @Override
            public void visitDirectory(FileSystemDirectory dir) {
                // don't add directories, only files
            }
        }
        final FileFilter filter = new FileFilter();
        while (iterator.hasNext()) {
            iterator.next()
                .accept(filter);
        }
        return filter.files;
    }

}
```

Ricordate che c'è un template "while – iterate with iterator".

In *FileFilter*, classe locale del metodo, potrebbe sembrare di aver violato un po' di principi: la variabile di istanza *files* NON è privata e ci si accede direttamente nel resto del metodo. Inoltre, usiamo un tipo concreto per dichiarare la variabile di istanza. Questo NON è assolutamente un problema in questo contesto specifico: quella classe è visibile e usabile solo all'interno di quel metodo quindi è sensato prendere queste scelte, che rendono addirittura il codice più leggibile. Tanto più che, come appena detto, la classe non è pensata per essere riusata al di fuori del metodo (anche volendo, non è accessibile al di fuori del metodo).

Il test:

```
@Test
public void testRecursiveDirectory() {
    FileSystemDirectory dir = new FileSystemDirectory("aDir");
    FileSystemFile file1 = new FileSystemFile("aFile1.txt");
    dir.add(file1);
    FileSystemDirectory nestedDir = new FileSystemDirectory("aNestedDir");
    dir.add(nestedDir);
    nestedDir.add(new FileSystemFile("aFile.txt"));
    FileSystemFile file2 = new FileSystemFile("aFile2.txt");
    dir.add(file2);
    assertThat(new DirectoryFilesCollectionSupplier(dir).get())
        .containsExactlyInAnyOrder(file1, file2);
}
```

Ovviamente se usate Java 11 (o comunque una versione di Java dalla 10 in poi), potete sfruttare il meccanismo di *var* visto a lezione (nella parte finale del pattern *Adapter*) e usare una classe anonima, invece di una classe locale al metodo:

```
    @Override
    public Collection<FileSystemFile> get() {
        Iterator<FileSystemResource> iterator = dir.iterator();
        // richiede almeno Java 10
        final var filter = new FileSystemVisitor() {
            ArrayList<FileSystemFile> files = new ArrayList<>();

            @Override
            public void visitFile(FileSystemFile file) {
                files.add(file);
            }

            @Override
            public void visitDirectory(FileSystemDirectory dir) {
                // don't add directories, only files
            }
        };
        while (iterator.hasNext()) {
            iterator.next()
                .accept(filter);
        }
        return filter.files;
    }
```

ATTENZIONE: se usate Java 8 la modifica sopra non compilerà perché "var" non è riconosciuto dal compilatore.

Volendo, si può comunque utilizzare una classe anonima anche in Java 8, purché si sposti la dichiarazione della lista per il risultato, "files", al di fuori della classe anonima:

```
    @Override
    public Collection<FileSystemFile> get() {
        Iterator<FileSystemResource> iterator = dir.iterator();
        ArrayList<FileSystemFile> files = new ArrayList<>();
        FileSystemVisitor filter = new FileSystemVisitor() {

            @Override
            public void visitFile(FileSystemFile file) {
                files.add(file);
            }

            @Override
            public void visitDirectory(FileSystemDirectory dir) {
                // don't add directories, only files
            }
        };
        while (iterator.hasNext()) {
            iterator.next()
                .accept(filter);
        }
        return files;
    }
```

NOTA: è considerata buona pratica di programmazione pulita mettere un commento che spiega perché l'implementazione di un metodo è vuota, come nel caso di *visitDirectory* appena mostrato.

6 Visitor Adapter

Implementare (nel package principale *mp.exercise.filesystem*) la classe astratta *FileSystemVisitorAdapter* che implementa l'interfaccia visitor con metodi *visit* di default: per i file non fa niente, per le directory visita ricorsivamente i contenuti. Al momento non è necessario testare questa classe. Dove e come possiamo usare questo adapter per rifattorizzare quello che abbiamo già implementato? (senza rompere i test ovviamente). Dove non conviene usarlo? Procedere alla rifattorizzazione in modo opportuno.

```
public abstract class FileSystemVisitorAdapter implements FileSystemVisitor {

    /**
     * The default implementation does nothing.
     */
    @Override
    public void visitFile(FileSystemFile file) {
        // does nothing
    }

    /**
     * The default implementation visit all the directory contents recursively.
     */
    @Override
    public void visitDirectory(FileSystemDirectory dir) {
        Iterator<FileSystemResource> iterator = dir.iterator();
        while (iterator.hasNext()) {
            iterator.next().accept(this);
        }
    }

}
```

Nella classe locale *FileFilter* di *DirectoryFilesCollectionSupplier* non conviene usare l'adapter come superclasse: l'implementazione di default di *visitDirectory* andrebbe in profondità mentre in questo caso vogliamo solo capire se si tratta di un singolo file o di una singola directory, quindi dovremmo comunque ridefinire *visitDirectory* in modo che non faccia niente (quindi sarebbe una sovrascrittura completa del metodo ereditato); il codice sarebbe praticamente uguale a quello che avevamo già scritto:

```
public class DirectoryFilesCollectionSupplier
             implements Supplier<Collection<FileSystemFile>> {
...
    @Override
    public Collection<FileSystemFile> get() {
        Iterator<FileSystemResource> iterator = dir.iterator();
        class FileFilter extends FileSystemVisitorAdapter {
            ArrayList<FileSystemFile> files = new ArrayList<>();

            @Override
            public void visitFile(FileSystemFile file) {
                files.add(file);
            }

            @Override
            public void visitDirectory(FileSystemDirectory dir) {
                // don't add directories, only files
            }
        }
...
```

In *FileSystemLsVisitor* possiamo riusare la visita ricorsiva dell'implementazione di *visitDirectory* dell'adapter; dobbiamo comunque ridefinire *visitDirectory* perché prima della visita ricorsiva vogliamo compiere qualche azione e poi chiamare *super*.

Quindi *FileSystemLsVisitor* diventa:

```
public class FileSystemLsVisitor extends FileSystemVisitorAdapter {

    private FileSystemPrinter printer;

    public FileSystemLsVisitor(FileSystemPrinter printer) {
        this.printer = printer;
    }

    @Override
    public void visitFile(FileSystemFile file) {
        printer.print("File: " + file.getName());
    }

    @Override
    public void visitDirectory(FileSystemDirectory dir) {
        printer.print("Directory: " + dir.getName());
        super.visitDirectory(dir);
    }

}
```

Allo stesso modo possiamo modificare *FileSystemNonRecursiveLsVisitor*.

Ovviamente rilanciamo i test per assicurarci che tutto continui a funzionare.

7 Utility methods

Implementare la classe *FileSystemUtils* che contiene solo metodi statici di utilità:

- *toFile* che, dato un *FileSystemResource*, restituisce un *Optional* con un *FileSystemFile* se la risorsa passata è un file o un *Optional* vuoto altrimenti
- *toDirectory*, che fa la stessa cosa ma per una una directory

Usare AssertJ per le asserzioni sugli *Optional* restituiti.

Una volta implementata e testata la classe *FileSystemUtils*, uno dei suoi metodi di utilità può essere usato per rifattorizzare una delle classi che abbiamo già implementato?

La classe può essere implementata come segue:

```
public class FileSystemUtils {

    private FileSystemUtils() {
        // Utility classes, which are collections of static members,
        // are not meant to be instantiated.
        // They should not have public constructors.
    }

    private static class FileFilter implements FileSystemVisitor {
        FileSystemFile file = null;

        @Override
        public void visitFile(FileSystemFile file) {
            this.file = file;
        }

        @Override
        public void visitDirectory(FileSystemDirectory dir) {
            // skip directories, only files
        }
    }

    public static Optional<FileSystemFile> toFile(FileSystemResource res) {
        final FileFilter filter = new FileFilter();
        res.accept(filter);
        return Optional.ofNullable(filter.file);
    }

    private static class DirectoryFilter extends FileSystemVisitorAdapter {
        FileSystemDirectory dir = null;

        @Override
        public void visitDirectory(FileSystemDirectory dir) {
            this.dir = dir;
        }
    }

    public static Optional<FileSystemDirectory> toDir(FileSystemResource res) {
        final DirectoryFilter filter = new DirectoryFilter();
        res.accept(filter);
        return Optional.ofNullable(filter.dir);
    }
}
```

Notare il commento nel costruttore: è buona norma non permettere di istanziare una classe che ha solo membri statici.

Usiamo AssertJ nei test:

```
public class FileSystemUtilsTest {

    @Test
    public void testToFile() {
        FileSystemDirectory dir = new FileSystemDirectory("aDir");
        FileSystemFile file = new FileSystemFile("aFile1.txt");
        dir.add(file);

        assertThat(FileSystemUtils.toFile(dir))
            .isEmpty();
        assertThat(FileSystemUtils.toFile(file))
            .isPresent()
            .containsSame(file);
    }

    @Test
    public void testToDir() {
        FileSystemDirectory dir = new FileSystemDirectory("aDir");
        FileSystemFile file = new FileSystemFile("aFile1.txt");
        dir.add(file);

        assertThat(FileSystemUtils.toDir(file))
            .isEmpty();
        assertThat(FileSystemUtils.toDir(dir))
            .isPresent()
            .containsSame(dir);
    }

}
```

Notare che per essere sicuri, la directory contiene un file: così ci assicuriamo che il visitor usato non vada in profondità. Notare inoltre l'uso delle asserzioni di AssertJ in presenza di *Optional*.

Possiamo usare questa classe di utilità in qualche classe esistente?

La classe *DirectoryFilesCollectionSupplier* adesso può essere semplificata come segue:

```
public class DirectoryFilesCollectionSupplier
             implements Supplier<Collection<FileSystemFile>> {

    private FileSystemDirectory dir;

    public DirectoryFilesCollectionSupplier(FileSystemDirectory dir) {
        this.dir = dir;
    }

    @Override
    public Collection<FileSystemFile> get() {
        Iterator<FileSystemResource> iterator = dir.iterator();
        ArrayList<FileSystemFile> files = new ArrayList<>();
        while (iterator.hasNext()) {
            FileSystemUtils.toFile(iterator.next())
                .ifPresent(files::add);
        }
        return files;
    }
```

```
}
```

Come sempre rilanciamo i test per esser sicuri di non aver “rotto” niente.

8 Conclusioni

Usando il pattern Visitor, abbiamo voluto rimuovere dalla gerarchia del file system la responsabilità di gestire operazioni come *ls* e altre operazioni di ricerca/query. Tuttavia, la gerarchia del file system rappresenta ancora una valida applicazione del pattern *Composite* in quanto rimane comunque nel tipo più astratto (*FileSystemResource*) l’operazione di “copia” che è implementata secondo lo schema del pattern nella foglia e nel composto. Inoltre, è sensato lasciare l’operazione di copia all’interno della gerarchia in quanto tale operazione richiede accesso a dettagli interni implementativi (ad es., come la directory si mantiene i file contenuti). Estrarre l’operazione di copia in un visitor invece non avrebbe molto senso, e saremmo costretti a esporre troppi dettagli della gerarchia del file system.