Relazione per “FileMiner”

Lorenzo Chiana, Michele Durante, Daniele Gambaletta

01 Marzo 2016

Capitolo 1

- **Analisi**

1.1 Requisiti

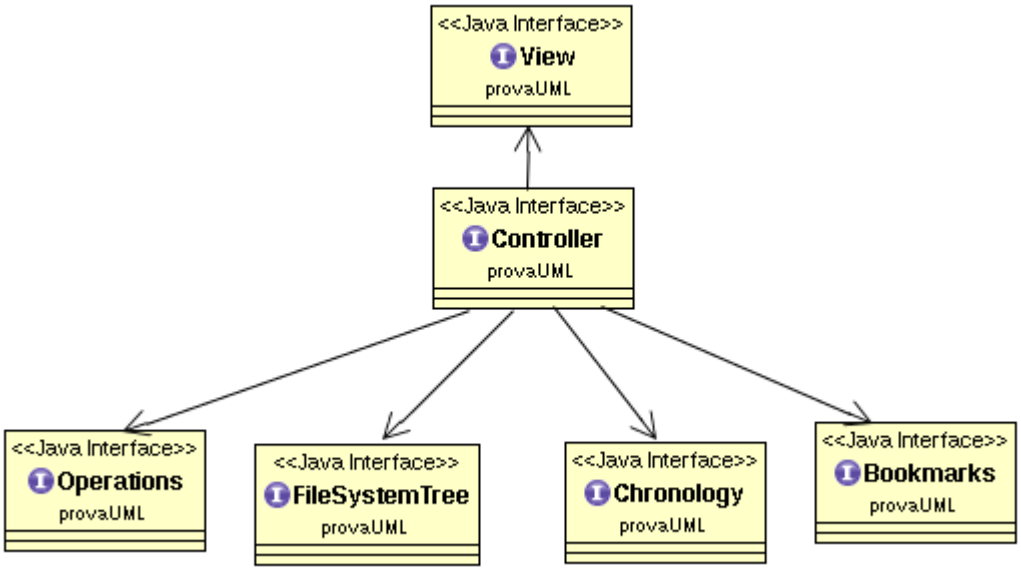
Il software è un file manager, un'applicazione in grado di gestire le operazioni basilari sul file system del sistema operativo ospite insieme a funzionalità aggiuntive. Queste ultime serviranno all'utente per permettergli un gradevole utilizzo del programma e un maggior controllo delle risorse a disposizione.
Il progetto nasce dall'idea di offrire agli utenti un software accessibile in modo gratuito ed open source.
Le caratteriste che il software dovrà avere sono le seguenti:

- Interfaccia grafica che permetterà di visualizzare la rappresentazione logica del file system e il suo contenuto.
- Gestione di file e directory:
  - Creazione;
  - Copia;
  - Incolla;
  - Spostamento;
  - Link;
  - Cancellazione;
  - Modifica.
- Visualizzazione dei permessi di accesso al file.
- Compressione e decompressione file.
- Cronologia delle directory visualizzate.
- Segnalibri.

1.2 Analisi e modello del dominio

FileMiner dovrà essere in grado di accedere ai vari nodi del file system della macchina su cui sta girando ed interagire con loro indipendentemente dal sistema operativo.
La difficoltà primaria sarà quella di riuscire a visualizzare ed interagire col file system ed immettere la sua struttura in una struttura dati apposita.
La seconda difficoltà si porrà al momento in cui si dovrà aggiornare la suddetta struttura dati ad ogni operazione tipo taglia, incolla, elimina… ecc. ecc.
Ultimo problema sarà gestire l'iterazione con l'utente.

Di seguito uno schema abbozzato che mostra le principali parti funzionali:

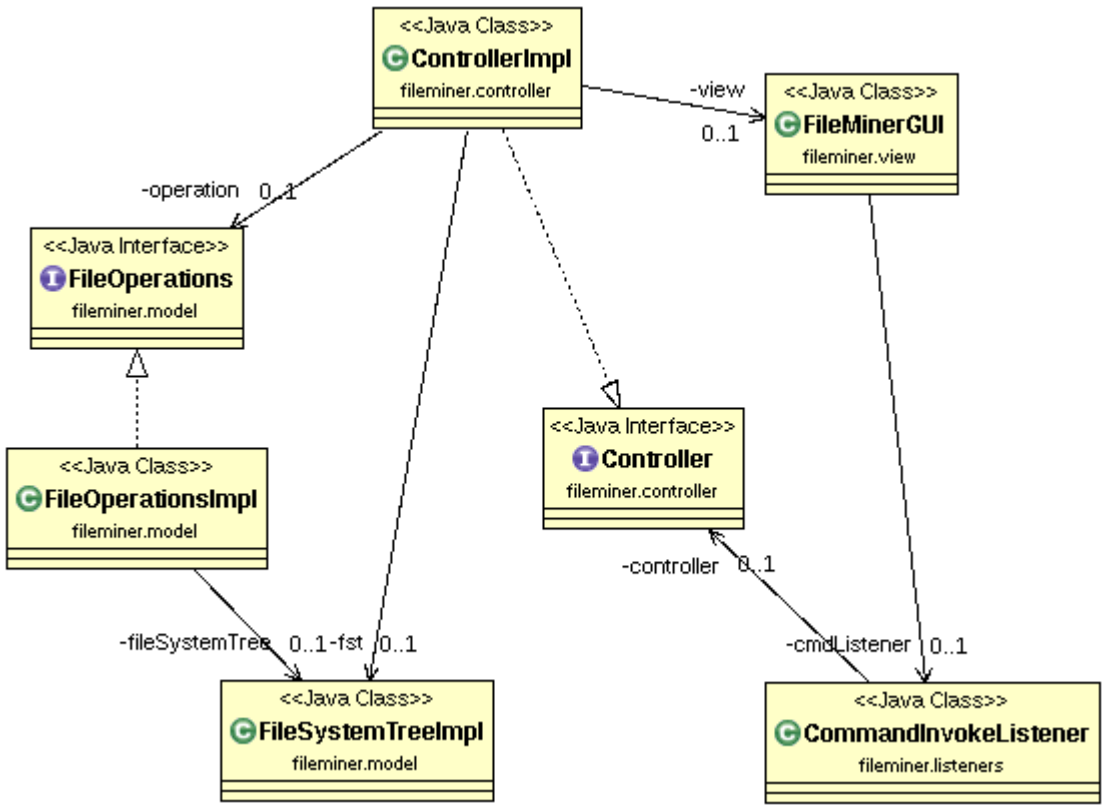


Capitolo 2

- **Design**

2.1 Architettura

Il software è stato organizzato seguendo il pattern MVC.
La classificazione in package rispecchia tale suddivisione.



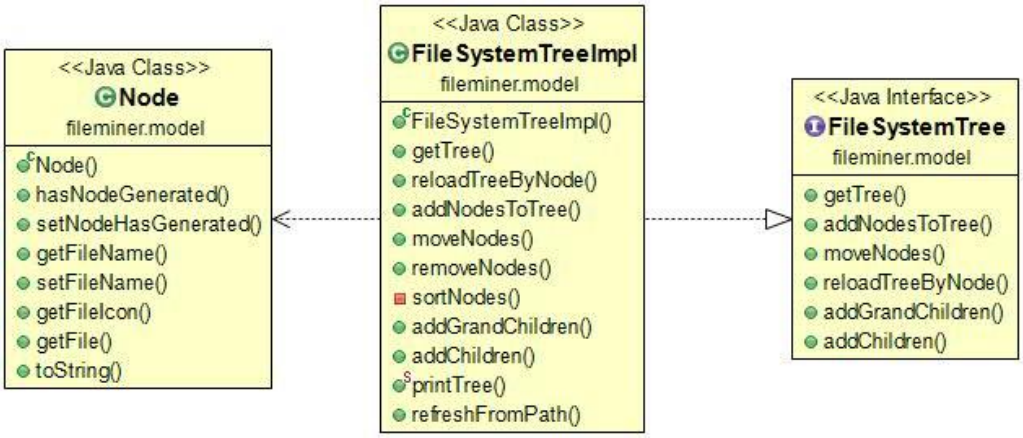
In tale suddivisione il controller acquisirà gli input dell'utente tramite la view e comanderà il model il quale si aggiornerà di conseguenza.
Le parti sono messe in contatto tra di loro tramite apposite interfacce, che permettono di gestire il progetto in modo modulare.
L'unica iterazione tra view e model in questo progetto è presente quando la view prende dal model i nodi dell'albero del file system per farli visualizzare all'utente.
Questa iterazione non viola il pattern architetturale MVC dato che la view può usare metodi del model a patto che quest'ultimo non venga aggiornato.

2.2 Design dettagliato

- **Model** (Gambaletta Daniele)

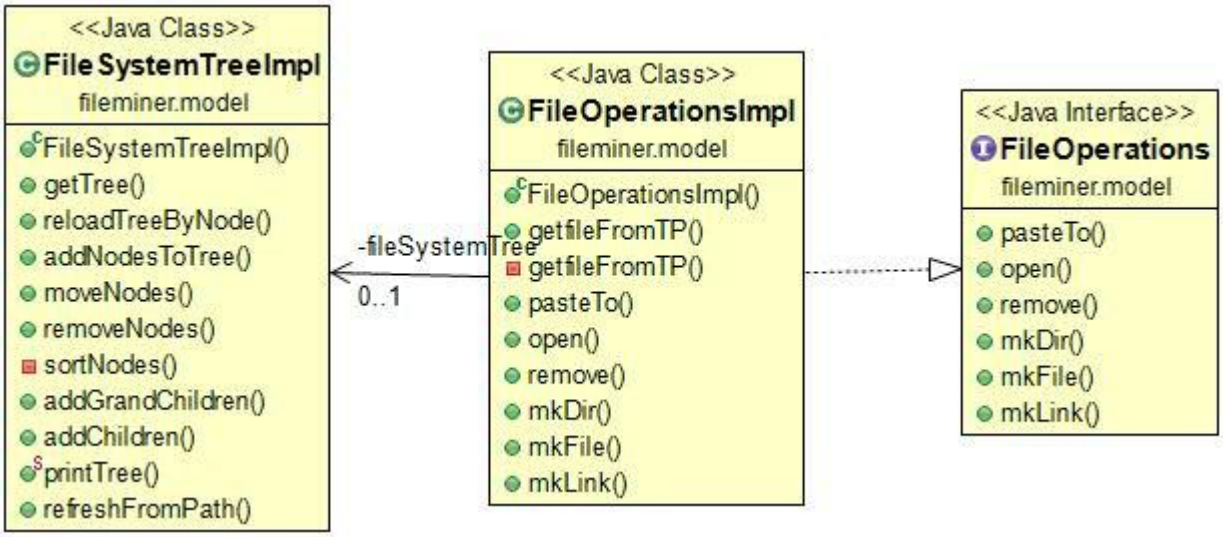
Nel model sono presenti tutte le funzionalità riguardo la gestione dell'albero del file system, la struttura della tabella dei files per la view, le operazioni base sui files(copia, taglia, incolla, comprimi, ecc...), compressione e decompressione in zip e la gestione dei bookmarks, utilizzando il factory pattern.

Per la gestione dell'albero del file system, è stata creata un'interfaccia con l'elenco dei metodi e una classe nella quale vengono implementati. Ogni nodo è l'istanza della classe Node nella quale vengono memorizzate, oltre al file, altre informazioni, per esempio, per sapere se quel nodo ha già generato nodi o deve ancora generarli nell'albero.

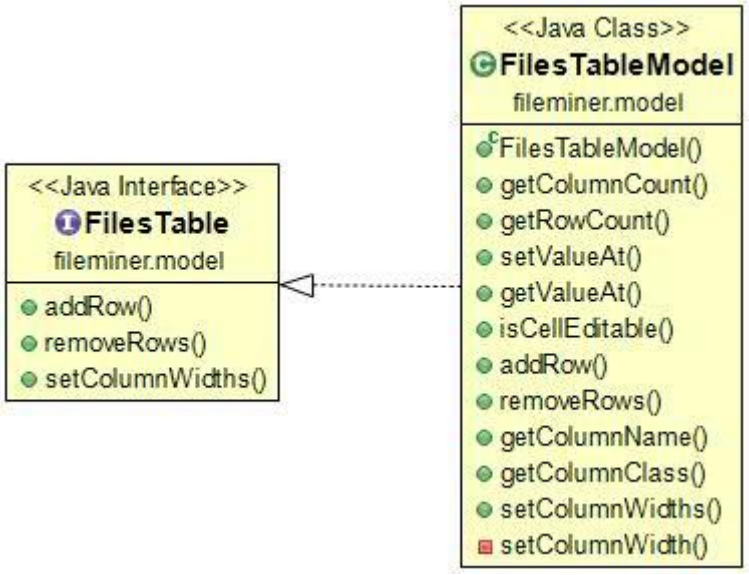


Per le operazioni sui file è stata sempre creata un'interfaccia con l'elenco di tutti i metodi relativi alla gestione dei file e alla loro creazione, con la relativa classe che li implementa. A questa classe viene anche passato,tramite costruttore, un FileSystemTreeImpl per poterlo aggiornare nel caso vengano fatti dei cambiamenti nel FileSystem. E' stato impiegato l'uso della libreria CommonsIO,

anche se utilizzata in piccola parte, ha aiutato molto nell'ottimizzare e gestire in modo efficiente le operazioni sui files.
Essendo questa classe estendibile c'è la possibilità, in futuro, di aggiungere altre operazioni sui files o modificarne l'implementazione.

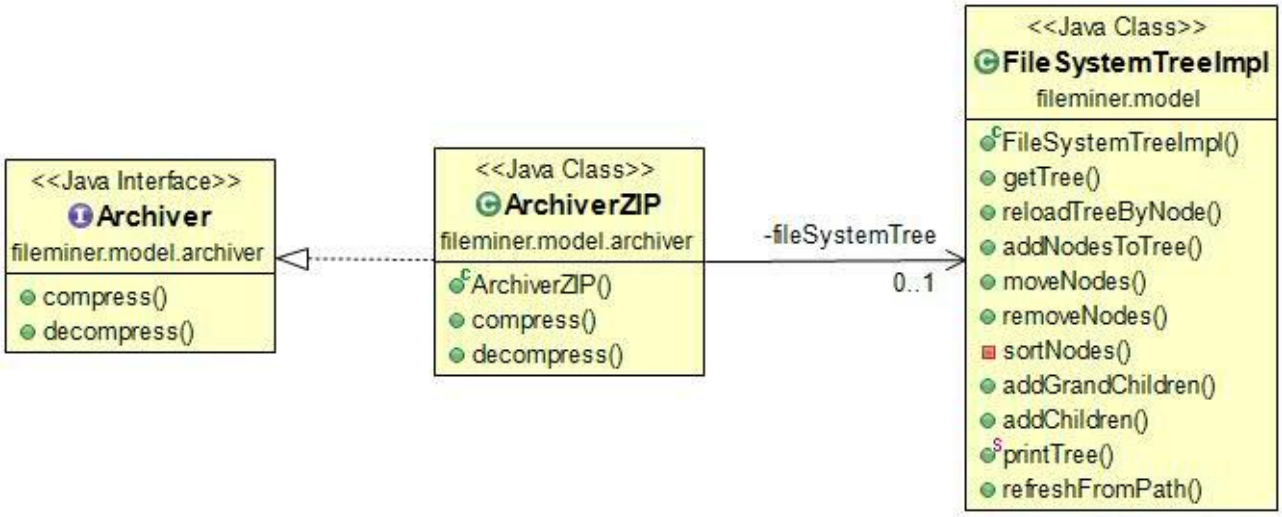


E' stata creata una interfaccia con relativa classe per la struttura della tabella nella view. In questo modo al'avvio dell'applicazione la view prenderà i dati da essa per poterne costruire una JTable.

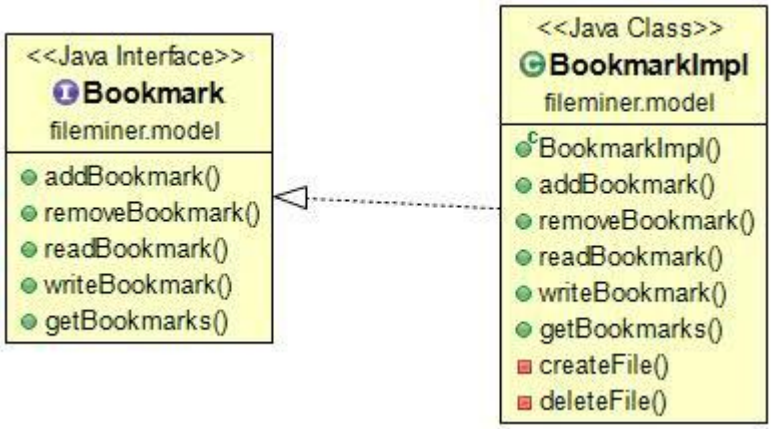


Per la compressione e la decompressione di file o cartelle è stata utilizzata la libreria Zip4j gestione di archivi zip.

Questa libreria non complessa ma efficiente, fornisce i metodi base per aggiungere o estrarre i files dall'archivio. Alla istanziazione di ArchiverZIP viene passato, tramite costruttore, anche il FileSystemTreeImpl, per aggiornare l'albero interno dell'applicazione.

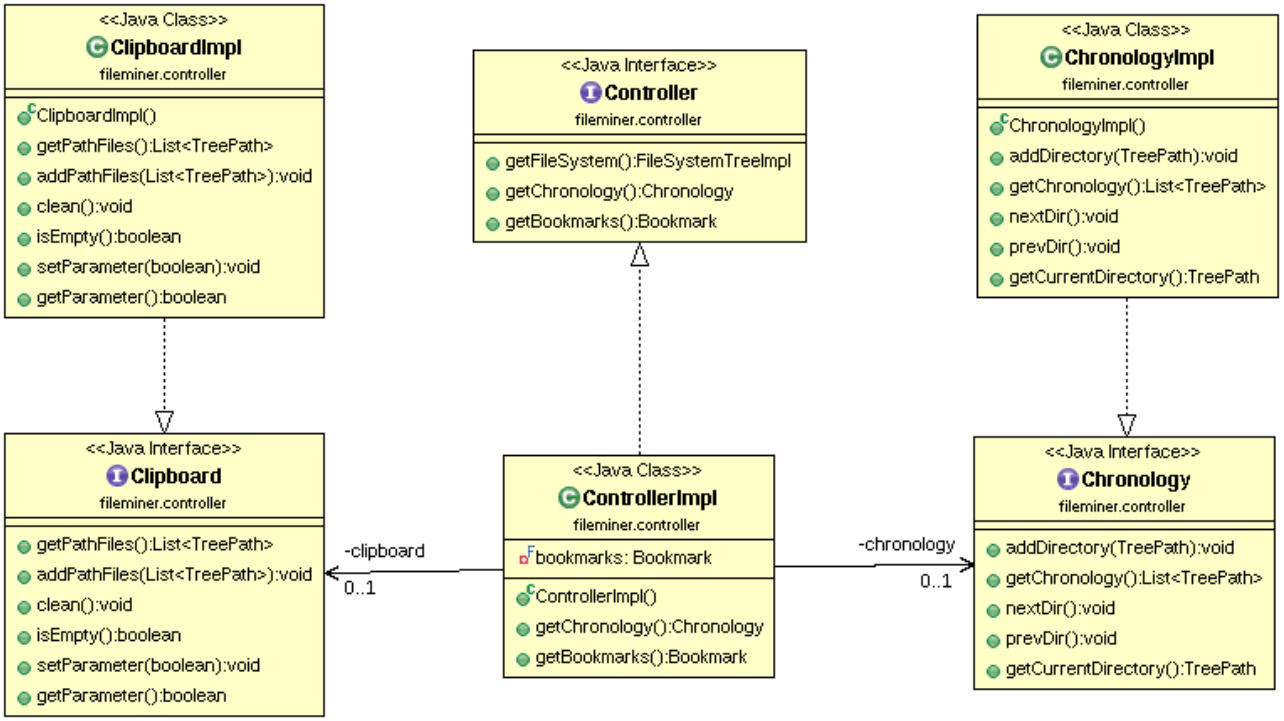


Per la classe BookmarkImpl, viene salvato un file, nella directory dell'utente, la quale contiene una lista di percorsi riguardanti i preferiti scelti da esso, grazie alla view. I metodi della classe servono per gestire e salvare i bookmark dell'utente in modo che una volta riaperta l'applicazione, vengano inizializzati.

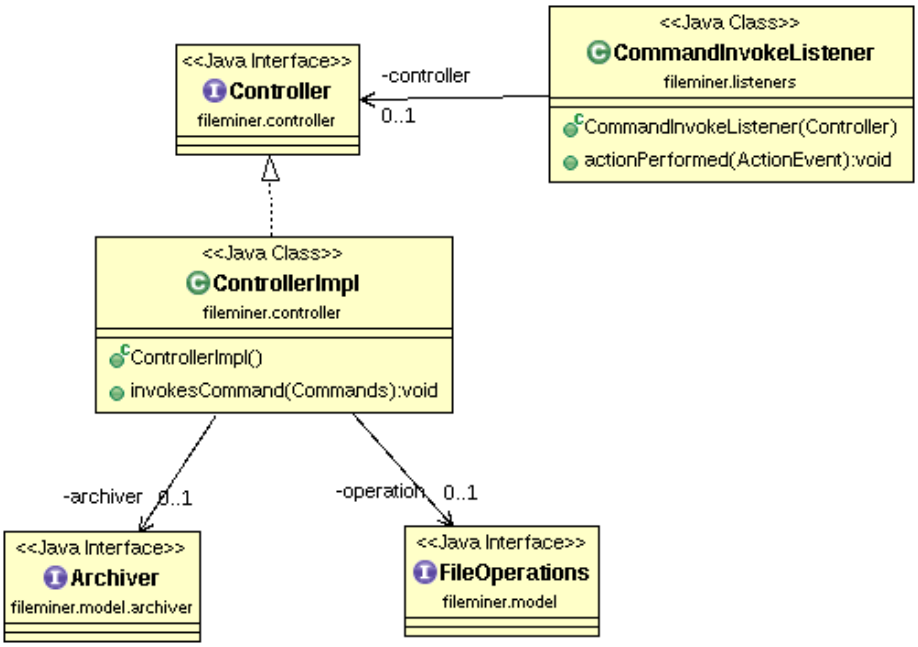


- **Controller** (Chiana Lorenzo)

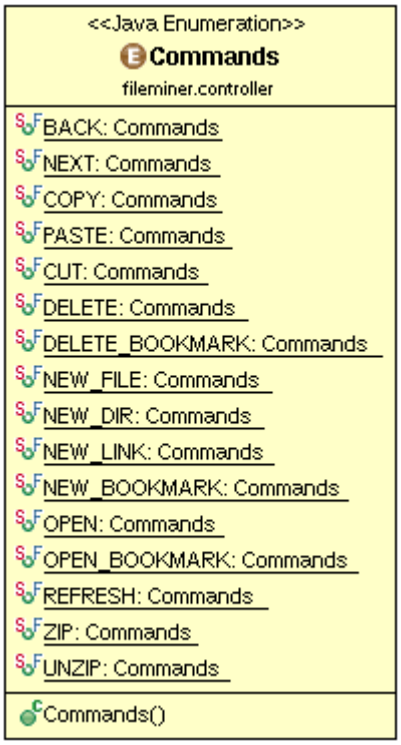
Al suo interno, il Controller è formato da altre sottoclassi atte alla gestione delle varie fasi dell'esecuzione dell'applicazione.
ControllerImpl gestisce le inizializzazioni varie.



In questo UML si nota come la classe ControllerImpl gestisce altre due sottoclassi: Chronology e Clipboard.
L'interfaccia Chronology e la sua relativa implementazione, ChronologyImpl, hanno lo scopo di tenere memorizzate le directory accedute dall'utente, come una vera e propria cronologia delle directory visitate in quel momento.
ControllerImpl fornisce il metodo “getChronology” per la gestione della classe relativa alla cronologia.
Clipboard, invece, viene utilizzata direttamente da ControllerImpl per la gestione delle operazioni di copia, incolla e taglia.
Infatti questa classe si occupa della memorizzazione in una lista dei vari nodi copiati, o tagliati, per poi recuperarli per l'operazione di incolla.
Inoltre ControllerImpl è richiamata dalla view attraverso il listener “CommandInvokeListener” qualora in essa ci sia un evento che richiede un'operazione di un comando implementato nel model.

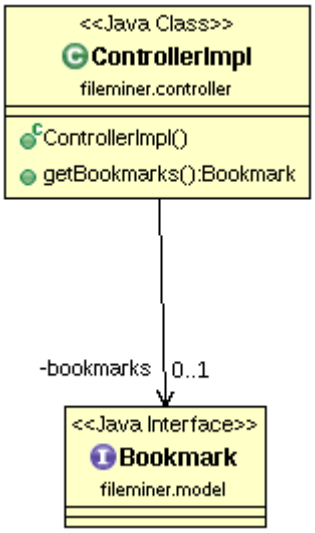


In questa classe è presente un metodo, invokesCommand, che a seconda del comando richiesto (copia, taglia, incolla, ecc. ecc.) invoca il metodo opportuno del model.
Per motivi di comodità e leggibilità i vari comandi messi a disposizione dal model li ho messi in una classe enum in modo tale da averli subito a disposizione richiamandola.

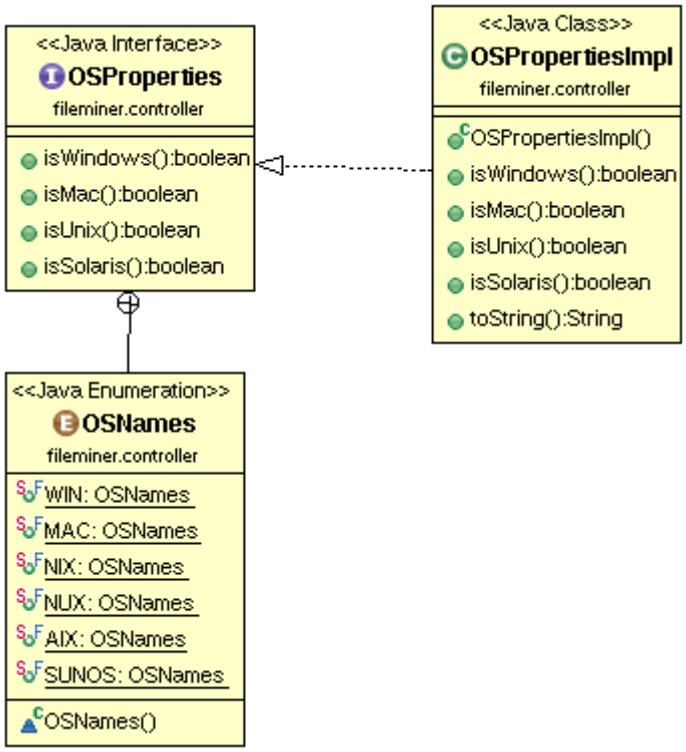


Un altro vantaggio di usare una classe enum per tenere i vari comandi utilizzabili dal programma è una possibile estensibilità del codice, inserendo qui i nuovi comandi che il model offre.

In questo metodo, inoltre, gestisco i vari errori ed eccezioni lanciati dalle operazioni del model e le faccio stampare come messaggio all'utente nella console del nostro applicativo (in basso).
Sempre in ControllerImpl vi è un metodo che richiama il model per la gestione dei bookmarks.



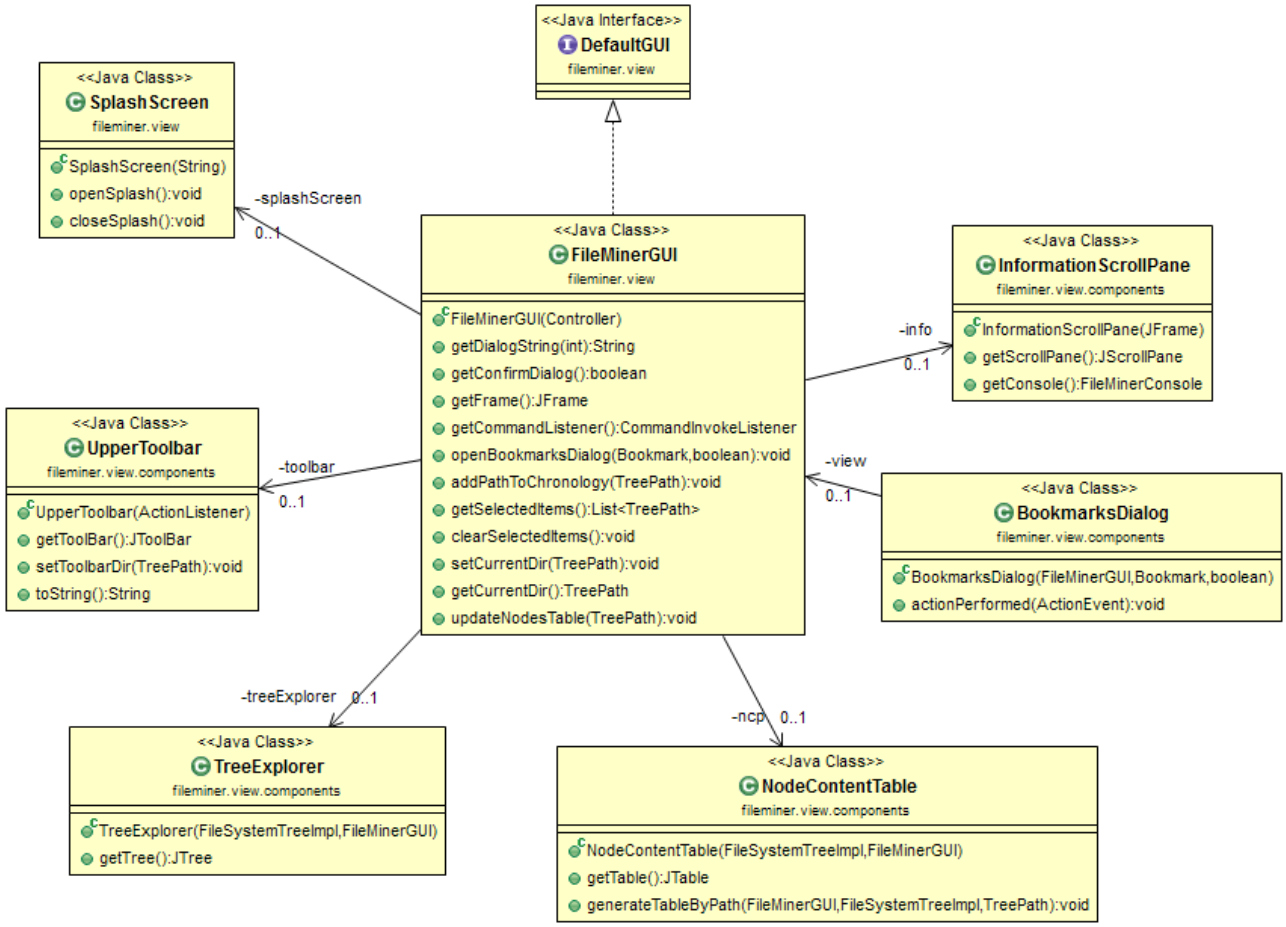
L'ultima classe usata nel controller è OSPropretiesImpl, la quale viene utilizzata dalla view per ottenere i dettagli del sistema operativo su cui l'applicazione sta girando.



Anche in questa classe utilizzo una enum per tenere l'elenco dei vari sistemi operativi, sempre per questioni di riusabilità, estensibilità e comodità.

Quest'ultima classe è stata realizzata prendendo spunto da:
http://www.mkyong.com/java/how-to-detect-os-in-java-systemgetpropertyosname/
Per l'implementazione del controller ho utilizzato il Factory Method come pattern di progettazione, creando classi attraverso l'implementazione di interfacce.

- **View** (Durante Michele)



La classe principale ***FileMinerGUI*** gestisce tutti i componenti visualizzati all'interno del programma. All'interno del suo costruttore, dopo esser stata inizializzata dal controller, viene creato uno splashscreen che avvisa l'utente del caricamento del programma, e subito dopo viene creato uno *SwingWorker* che effettua le operazioni d'inizializzazione dei componenti (compreso il main frame).
Quest'ultimo si è rivelato molto utile per la corretta gestione dell'Event-Dispatch thread, in modo da non bloccare la reattività dell'interfaccia grafica sia durante il caricamento dell'applicazione che all'avvio di un comando da parte dell'utente, poiché permette anche operazioni di finalizzazione.

I componenti e le classi sono:

- All'interno del main frame viene inizializzato in un metodo privato il componente *JMenuBar* con tutti gli elementi necessari per eseguire buona parte dei comandi implementati dall'applicazione.

- ***SplashScreen***: classe che estende una *JWindow* avviata prima di iniziare il caricamento dei componenti del frame principale per indicare all'utente che l'applicazione si sta inizializzando. Viene chiusa dal main frame quando il caricamento dell'applicazione è terminato.

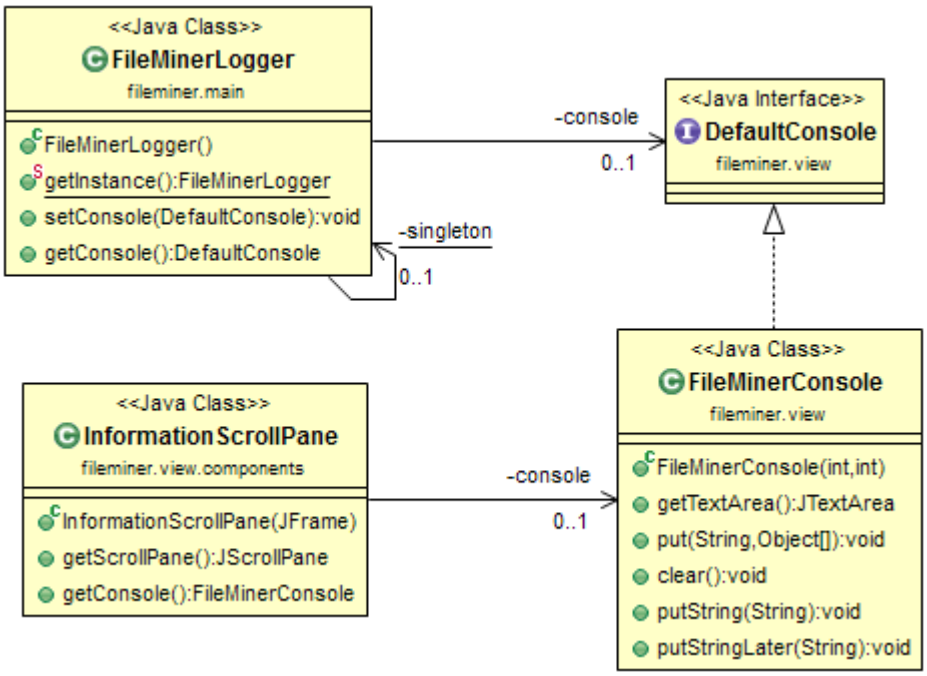
- ***UpperToolbar***: classe che incorpora una *JToolbar* contenente due frecce per la navigazione della cronologia, una funzione refresh per l'attuale directory e la visualizzazione della directory corrente.

- ***TreeExplorer***: classe che incorpora un *JTree* corrispondente alla rappresentazione grafica del modello di albero generato dalla classe *FileSystemTreeImpl* passatogli dal controller.

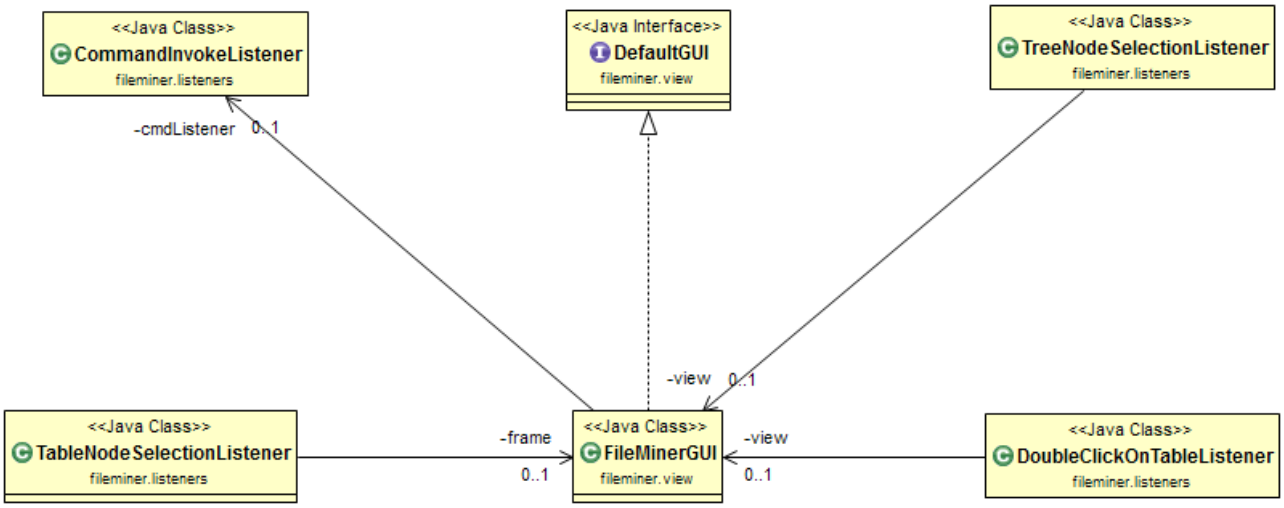
- ***NodeContentTable***: classe che incorpora una *JTable* in grado di rappresentare tramite un modello di tabella creato dal model le informazioni circa i file e le directory che sono contenute nell'attuale directory corrente.

- ***BookmarksDialog***: classe che, quando viene inizializzata, crea una *JDialog* che rimane attiva finché l'utente non decide di chiuderla e che gestisce l'eliminazione e l'apertura dei segnalibri.

- ***InformationPane***: classe che incorpora un *JScrollPane* e la classe *FileMinerConsole*, quest'ultima verrà poi utilizzata come mezzo di comunicazione con l'utente.



Per comunicare all'utente i vari messaggi generati dall'applicazione ho deciso di creare una classe di nome ***FileMinerLogger*** che utilizza il paradigma di programmazione "singleton", ovvero è disponibile una sola istanza di questa classe per tutta la durata dell'applicazione. Essa consente di settare all'avvio una qualsiasi classe che permette la visualizzazione di informazioni testuali implementando un'interfaccia *DefaultConsole*, per esempio *FileMinerConsole* viene settata come "console" di riferimento (al suo interno viene gestita una *JTextArea*).
Per effettuare un log è sufficiente richiamare la classe *FileMinerLogger*, recuperare l'istanza e la console affibbiata e utilizzare i metodi forniti per stampare in fondo all'applicazione.



Per i listener si fa principalmente riferimento ai listener dei vari componenti Swing, in particolare:

- ***CommandInvokeListener*** implementa un *ActionListener* per tutti i *JButton* e i *JMenuItem* che ne fanno uso per richiamare i comandi dal controller. Implementa uno SwingWorker in grado di lasciare la GUI reattiva e di permettere
- ***TreeNodeSelectionListener*** implementa un *TreeSelectionListener* per il componente *JTree* che gestisce l'update dei nodi quando l'utente ne seleziona uno dall'albero.
- ***TreeNodeExpandListener*** implementa un *TreeWillExpandListener* per il componente *JTree* che gestisce l'espansione/collasso dei nodi dell'albero.
- ***TableNodeSelectionListener*** implementa un *TableModelListener* per il componente *JTable* che gestisce la selezione del file o della directory attraverso la spunta implementata nella prima colonna.
- ***DoubleClickOnTableListener*** estende *MouseAdapter* per implementare la funzione che consente l'apertura in-app di una directory.

Per l'aspetto finale ho preso come spunto le diverse implementazioni di file manager dei sistemi operativi più conosciuti. Ho scelto di implementare solo due tipi di Look&Feel utilizzabili dall'utente poiché una più vasta disponibilità è stata considerata superflua da quella fornita dal sistema di default.
Il primo pattern scelto per la view è stato di tipo Adapter, poi successivamente ho deciso di lasciare che i componenti principale della GUI vengano gestiti da classi ben definite in grado di restituire il componente stesso e di operare su di esso.

Capitolo 3

- **Sviluppo**

3.1 Testing Automatizzato

E' stata creata una classe Test, che utilizza la libreria JUnit per provare alcuni metodi del model utilizzando gli assert.
Questo metodo di test esegue le seguenti operazioni base: crea una cartella, crea un file all'interno della cartella e elimina tutta la cartella con il contenuto.

3.2 Metodologia di lavoro

Chiana Lorenzo
Il mio ruolo principale è stata quella relativa al controller, gestendo le richieste della view con i relativi metodi del model.
Ho contribuito inoltre della risoluzione di alcuni semplici bug all'interno di alcuni metodi del model incontrati durante l'"interfacciamento" tra view e model.

Mi sono occupato inoltre delle varie stesure delle bozze della relazione ed analisi.
Complessivamente sono state dedicate alla parte di analisi del progetto 4/5 ore nel mese di dicembre insieme ai colleghi Gambaletta e Durante.
Queste ore di analisi comprendono: scelta del progetto, divisione dei compiti e struttura base del progetto.
Alla parte di implementazione e studio sono state dedicate 2 settimane di lavoro: dalle 5 alle 6 ore di lavoro per la prima settimana e 7 ore durante la seconda.
Inoltre dato il procedere a rilento durante la seconda settimana abbiamo sfruttato la giornata di venerdì 26 febbraio per una sessione di programmazione di 13 ore.
Nell'ultima settimana queste ore sono state impiegate anche per la ricerca di bugs e per la loro risoluzione insieme al gruppo.
Io e il mio gruppo abbiamo fatto uso massiccio di del DVCS e ci siamo coordinati al fine di riuscire ad ottenere un programma funzionante entro i termini previsti.
La mia parte ha interessato:

- Controller
- Gestione errori
- Gestione della cronologia
- Gestione della clipboard
- Check del sistema operativo

Gambaletta Daniele
Il mio ruolo principale è stato quello del model, gestendo la parte logica del programma.
Nel mese di dicembre tutto il gruppo ha utilizzato 4/5 ore per la stesura della prima analisi.
Inoltre per il progetto ho occupato dalle 6 alle 8 ore di lavoro delle ultime due settimane e mezzo di febbraio, con un picco di 13 ore il 26 febbraio insieme agli altri membri del gruppo.
Queste ore sono state impiegate, oltre per l'implementazione, per la ricerca e studio di librerie che mi permettessero di utilizzare i comandi base (copia, incolla, ecc. ecc.) e l'uso dell'archiviazione.
Io e il mio gruppo abbiamo fatto uso massiccio di del DVCS e ci siamo coordinati al fine di riuscire ad ottenere un programma funzionante entro i termini previsti.
La mia parte ha interessato:

- Model
- Creazione e gestione dell'albero delle directory
- Gestione dei bookmarks
- Gestione delle librerie Zip4j e CommonsIO
- Gestione archiviazione

- Implementazione comandi base come: copia, incolla, taglia ecc. ecc.

Durante Michele
Il mio ruolo principale è stato quello della view e di coordinatore del gruppo.
Ho contribuito alla risoluzione di vari bug nel model e nella risoluzione del problema di aggiornamento nodi che abbiamo incontrato durante l'implementazione.
Nel mese di dicembre tutto il gruppo ha utilizzato 4/5 ore per la stesura della prima analisi.
Per il progetto ho occupato dalle 7 alle 10 ore giornaliere per due settimane e mezzo e 13 ore il 26 febbraio.
Queste ore sono state impiegate per la realizzazione dell'interfaccia grafica, risoluzione di certi bug e risoluzione del problema di aggiornamento dei nodi.
Io e il mio gruppo abbiamo fatto uso massiccio di del DVCS e ci siamo coordinati al fine di riuscire ad ottenere un programma funzionante entro i termini previsti.
La mia parte ha interessato:

- View
- Creazione interfaccia grafica
- Creazione listener
- Risoluzione di diversi problemi
- Coordinamento dei membri del gruppo

Capitolo 4

- **Commenti finali**

4.1 Autovalutazioni e lavori futuri

Chiana Lorenzo
Sono particolarmente soddisfatto del risultato ottenuto, anche se ci siamo ridotti un po' troppo all'ultimo per l'implementazione causa esami.
La buona cooperazione e collaborazione del gruppo col resto del team ha aiutato nello sviluppo e nel raggiungimento degli obbiettivi del progetto.
La speranza per il futuro è quella di migliorare la stesura del codice e organizzare meglio il tempo a disposizione.
Gambaletta Daniele
Sono soddisfatto del risultato ottenuto.
A causa esami, ahimè, ci siamo ridotti alle ultime due settimane e mezzo per l'implementazione del progetto, però tutto sommato siamo riusciti a dare il massimo per il progetto e a finirlo in tempo, anche grazie ad una buona coordinazione del gruppo.

La speranza per il futuro è gestire meglio le tempistiche e migliorare l'utilizzo di pattern di progettazione.
Durante Michele
Tutto sommato sono anch'io soddisfatto del risultato ottenuto.
Come anticipato dai miei colleghi l'unica pecca è stata la gestione del tempo.
Per il futuro la mia speranza è quella di riuscire meglio nella stesura del codice e migliorare nell'utilizzo dei pattern di progettazione.

4.2 Fonti

Per la realizzazione di questo progetto abbiamo utilizzato come fonti:

- StackOverflow

Abbiamo preso parti di codice da questo sito: http://java-articles.info/articles/?p=637, e le abbiamo adeguate alle nostre necessità.

4.3 Bugs e difetti

Bugs:

- Il sistema operativo non fa creare il simboli link perché richide dei privilegi.
- La decompressione non aggiorna la tabella dei files (cosa risolvibile run-time premendo il bottone “refresh”);

Difetti:

- Quando si esegue uno sposta, incolla o una creazione, il file generato sovrascrive il precedente se con lo stesso nome.

Appendice A

- **Guida utente**

All’avvio dell’applicazione si presenterà una schermata di caricamento che segnala che il programma sta caricando il file system.
In alto vi sarà presente un menù con le varie operazioni effettuabili.
Al centro sul lato sinistro vi sarà la rappresentazione del file system con partizioni, directory e sottodirectory.
Sarà possibile interagire con entrando nelle varie directory.
Selezionata una directory al centro sul lato destro viene visualizzato il suo contenuto (sotto directory e file) e le relative informazioni.
In basso vi è presente una console dove vengono visualizzati i vari errori, l’inizio e la fine delle operazioni e la cronologia degli accessi alle directory.