Sapienza

Università di Roma

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

anno accademico 2008-09

Relazione finale su progetto interno

Realizzazione di un'applicazione per la visualizzazione grafica didattica e di debug per algoritmi in Java

Candidato:
Pier Paolo Ciarravano
Matr. 773970

Relatore:
Prof. Fabrizio d'Amore

A Renato, Graziella, Alessandra e Alessandro

che in questi anni hanno sempre creduto in me

Indice

Introduzione

La corretta programmazione degli algoritmi e delle strutture dati è fondamentale per l'efficienza dell'esecuzione di un programma.

Per permettere di analizzare complessi algoritmi, l'approccio visuale, attraverso la rappresentazione animata delle entità che entrano in gioco nell'esecuzione, semplifica lo studio, la comprensione e il debug del codice scritto. In questo ambito la ricerca, conosciuta con il nome di *Data Structure and Algorithm Visualisation*, è iniziata già a partire dai primi anni ottanta.

In questa relazione viene presentata un'applicazione, chiamata "*AlgoExplorer*", che permette la visualizzazione grafica animata delle strutture dati che entrano in gioco nell'esecuzione di un programma in linguaggio Java. Tale applicazione è stata sviluppata nell'ambito dello stage finale interno per il completamento della laurea di primo livello in Ingegneria Informatica.

L'applicazione offre un approccio didattico alla comprensione del funzionamento degli algoritmi, permettendo di visualizzare come gli oggetti, manipolati dagli algoritmi presi in esame, interagiscono tra loro e come sono relazionati e collegati. Inoltre l'applicazione in

esame permette un facile debug visuale degli algoritmi, permettendo passo per passo, di visualizzare se le strutture dati utilizzate e le variabili realizzano il compito definito dal programma.

Obiettivi e motivazioni

Le rappresentazioni visuali sono molto utili per comprendere i meccanismi e il funzionamento di molte tipologie di algoritmi così da poterne più velocemente capire il comportamento.

E' universalmente riconosciuto che la visualizzazione grafica delle strutture dati e degli algoritmi è una potente alternativa alla descrizione testuale degli algoritmi o alla spiegazione verbale con l'ausilio di illustrazioni: la visualizzazione, specie se animata, migliora la comprensione e la dove siano sviluppati sistemi che permettono l'interazione nelle varie fasi di esecuzione degli algoritmi, è capace di catturare ancor meglio l'attenzione dei fruitori potenziando così lo scopo didattico.

Lo studio e la sperimentazione della visualizzazione delle strutture dati e degli algoritmi in maniera grafica ha inizio già a partire dal 1981 con l'animazione "Sorting out Sorting"[A1,B1] creata da Ronald Baeker; successivamente nel 1984 venne sviluppato BALSA[A2] il primo sistema per creare animazioni di algoritmi; attualmente esistono numerose animazioni di algoritmi ben conosciuti[B2] e molti sistemi per la visualizzazione runtime, molti

delle quali richiedono che i programmi scritti per essere visualizzati siano sviluppati seguendo delle particolari regole o implementando alcune librerie.

Parallelamente alla ricerca e allo sviluppo di sistemi per la visualizzazione di strutture dati e algoritmi, sono nati sistemi che, oltre a visualizzare graficamente l'esecuzione dei programmi, permettono anche il "*Debugging Visuale*"; questi sistemi pertanto aggiungono all'ausilio didattico, la capacità di ispezionare le variabili in diversi istanti dell'esecuzione e di capire se i valori delle variabili e il comportamento degli algoritmi allo studio coincidono con i valori attesi.

L'obiettivo dell'applicazione in esame, che da qui in poi chiameremo *AlgoExplorer*, nome scelto per evidenziare la capacità di "esplorare" il funzionamento di un algoritmo, è quello di visualizzare in maniera grafica e intuitiva, e non semplicemente in modo tabellare come la maggior parte delle applicazioni di debug, le strutture dati che entrano in gioco nell'esecuzione di un programma. Più precisamente si vuole visualizzare in maniera grafica, come le istanze delle classi di interesse utilizzate da un determinato programma, sono relazionate e collegate tra loro in istanti di esecuzione prefissati.

In numerosi testi didattici di fondamenti di informatica le istanze di classi sono spesso rappresentate come rettangoli contenenti dei valori che rappresentano gli attributi dell'istanza stessa, e con frecce

che rappresentano puntatori o variabili che referenziano altre istanze; in questo modo, ad esempio, una lista collegata viene spesso rappresentata come una lunga catena di rettangoli collegati insieme da frecce, in maniera sequenziale. AlgoExplorer ricrea questa metodologia di visualizzazione, assegnando così all'applicazione oltre che una funzione di debug funzionale anche una funzione didattica, perché permette di raffigurare le strutture dati di interesse nella maniera in cui siamo abituati ad immaginare oggetti e puntatori.

L'applicazione è sviluppata in linguaggio Java ed è capace di analizzare programmi scritti in questo stesso linguaggio.

I metodi per la visualizzazione degli algoritmi e delle strutture dati si possono suddividere in due tipologie:

1) metodi che attraverso l'esecuzione di un programma opportunamente configurato generano un'animazione non interattiva del comportamento dell'algoritmo; in questo caso l'animazione può anche essere esportata e visualizzata con un visualizzatore, che si limita esclusivamente a mostrare l'output dell'animazione, indipendentemente dal sistema che l'ha creata;
2) metodi che eseguono l'algoritmo allo studio e sono capaci in realtime di visualizzarne il comportamento, permettendo all'utente finale che visualizza l'animazione, la dove sia

richiesto e possibile, di interagire con i diversi valori in input all'algoritmo.

Entrambi questi metodi richiedono spesso un'implementazione preliminare del codice dell'algoritmo che rispetti alcune regole dell'applicazione che si occupa della visualizzazione, come già spiegato in precedenza; AlgoExplorer è un'applicazione che appartiene alla seconda tipologia descritta ed ha come obiettivo fondamentale quello di visualizzare il comportamento di un programma senza però che lo sviluppatore degli algoritmi conosca o implementi nessuna particolare regola per la creazione dell'algoritmo stesso o usi istruzioni e librerie esterne oltre a quelle di utilità per l'algoritmo.

AlgoExplorer in questo modo può essere utilizzata sia come applicazione per il debug di programmi Java sia semplicemente a scopo didattico per lo studio di algoritmi ben conosciuti in letteratura, permettendo l'interazione con diversi input in modo da facilitarne la comprensione.

Requisiti utente e specifiche funzionali

Si vuole realizzare un'applicazione per il linguaggio Java che permetta di visualizzare in maniera grafica lo stato delle variabili e degli oggetti gestiti e manipolati da algoritmi.
I seguenti esempi mostrano, con alcune ipotesi di applicazione, quali dovranno essere le visualizzazioni possibili grazie ad AlgoExplorer:

Esempio 1:
Creato un algoritmo che opera un ordinamento su una lista collegata, si vuole visualizzare come l'algoritmo opera sugli elementi della lista, e come gli elementi della lista sono relazionati tra loro ad istanti sequenziali dell'esecuzione;

Esempio 2:
Creato un algoritmo che gestisce alberi AVL, si vuole visualizzare come l'algoritmo opera sui nodi e come avvengono le eventuali rotazioni all'inserimento o alla cancellazione di un nodo;

Esempio 3:

Creato un programma che implementa l'algoritmo di Dijkstra su un grafo, si vuole visualizzare il grafo in maniera grafica e ad ogni interazione significativa dell'algoritmo si vuole visualizzare le scelte fatte dall'algoritmo stesso sul grafo.

Requisito fondamentale è che gli algoritmi presi in esame dall'applicazione non siano scritti specificatamente utilizzando regole o API dell'applicazione di visualizzazione; lo sviluppatore degli algoritmi che verranno analizzati dall'applicazione non dovrà pertanto conoscere o implementare nessuna particolare regola di creazione dell'algoritmo o usare istruzioni e librerie esterne oltre quelle di utilità per l'algoritmo stesso.

L'applicazione dovrà anche permettere la visualizzazione delle relazioni tra tutti gli oggetti e le variabili degli algoritmi in punti di esecuzione scelti sul codice sorgente, in modo da permettere l'esame dei valori delle variabili scelte e l'esame delle relazioni tra gli oggetti delle classi che si vogliono visualizzare.

L'applicazione dovrà permettere di operare lo studio degli algoritmi utilizzando una GUI (Graphical user interface) usabile e semplice.

La portabilità dell'applicazione AlgoExplorer sarà garantita dall'implementazione in Java SE 5 (Sun JDK 1.5 o versioni successive)[B3], senza dover utilizzare librerie native; pertanto

AlgoExplorer potrà essere eseguito su qualsiasi sistema operativo per cui si stata rilasciata questa versione di Java.

Descrizione delle operazioni desiderate

AlgoExplorer sarà costituito da due distinte fasi di interazione con l'utente: la prima fase sarà costituita dall'importazione delle classi che costituiscono il programma che si vuole eseguire, dalla scelta delle classi visualizzabili e dalla scelta delle rispettive variabili da ispezionare; la seconda fase sarà costituita dal controllo del programma che si vuole eseguire e della visualizzazione delle strutture dati che sono state scelte per la visualizzazione.

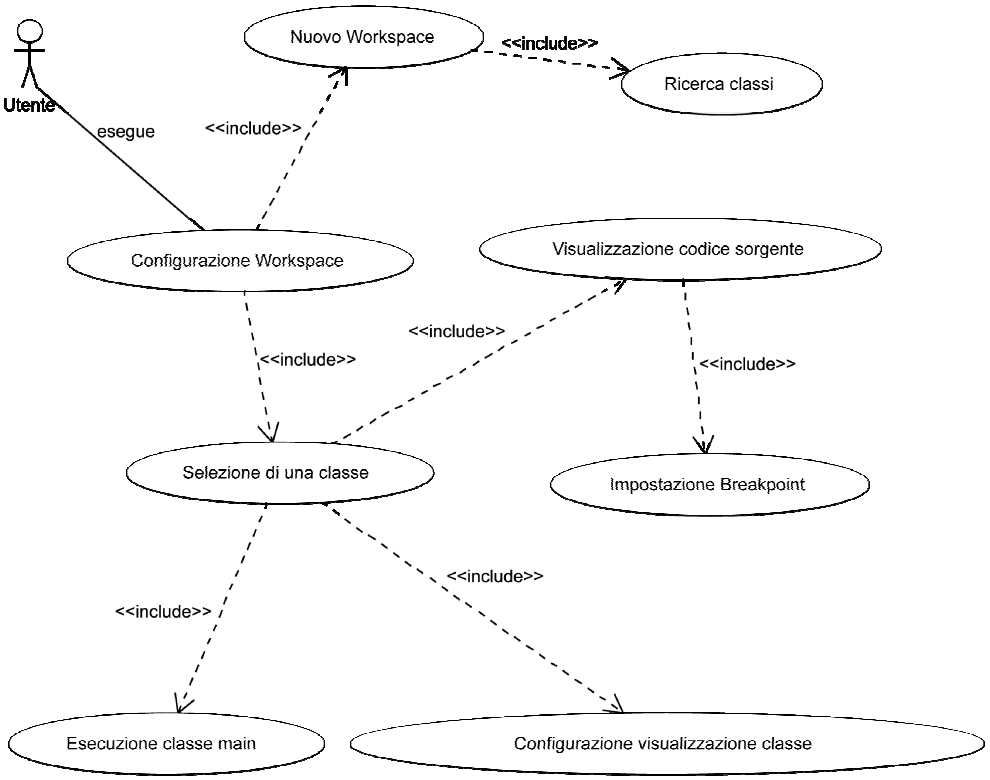
Nella prima fase, AlgoExplorer dovrà permettere l'importazione delle classi binarie e dei sorgenti Java (questi ultimi opzionali). Le classi importate saranno visualizzate in una struttura ad albero che ne evidenzia i package; le classi che contengono un metodo *main* saranno evidenziate in modo diverso, così da poter essere scelte per l'esecuzione. Per ogni classe sarà possibile visualizzare il codice sorgente se importato, impostare eventuali breakpoint in cui si vuole che AlgoExoplorer visualizzi lo stato degli oggetti e delle variabili, scegliere se le istanze della classe saranno visualizzate e quali dei propri attributi costituirà l'etichetta dell'istanza e scegliere gli attributi che si potranno ispezionare per conoscerne il valore nei diversi breakpoint individuati sul codice sorgente. Nel caso non sia stato inserito nessun breakpoint all'interno del codice sorgente, la

visualizzazione dello stato delle strutture dati scelte avverrà solamente al termine del metodo *main* del programma lanciato.
Una volta effettuate le scelte di visualizzazione desiderate si può passare all'esecuzione di una delle classi che contiene il metodo *main*. Sarà possibile impostare, per ogni programma che si vuole eseguire, eventuali parametri applicativi da passare al metodo *main* e eventuali opzioni per la Java Virtual Machine che lo eseguirà.
Nella seconda fase, AlgoExplorer permetterà la visualizzazione delle strutture dati scelte e il controllo del programma. In un apposita area dell'interfaccia utente, saranno visualizzate le istanze delle classi scelte e sarà possibile scegliere la scala di visualizzazione e il layout automatico che sarà usato per visualizzare le strutture dati (es: albero o grafo). Tramite appositi comandi, sarà inoltre possibile controllare l'esecuzione: inizialmente un programma sarà in sospensione e tramite un apposito comando potrà essere avviato; ad ogni breakpoint impostato sul codice sorgente, l'esecuzione si arresterà per visualizzare lo stato delle istanze delle classi selezionate per la visualizzazione e, quando desiderato, si potrà far proseguire l'esecuzione. Quando l'esecuzione è sospesa, sarà possibile cliccare le istanze visualizzate per ispezionare i valori degli attributi selezionati come ispezionabili nella prima fase di AlgoExplorer.

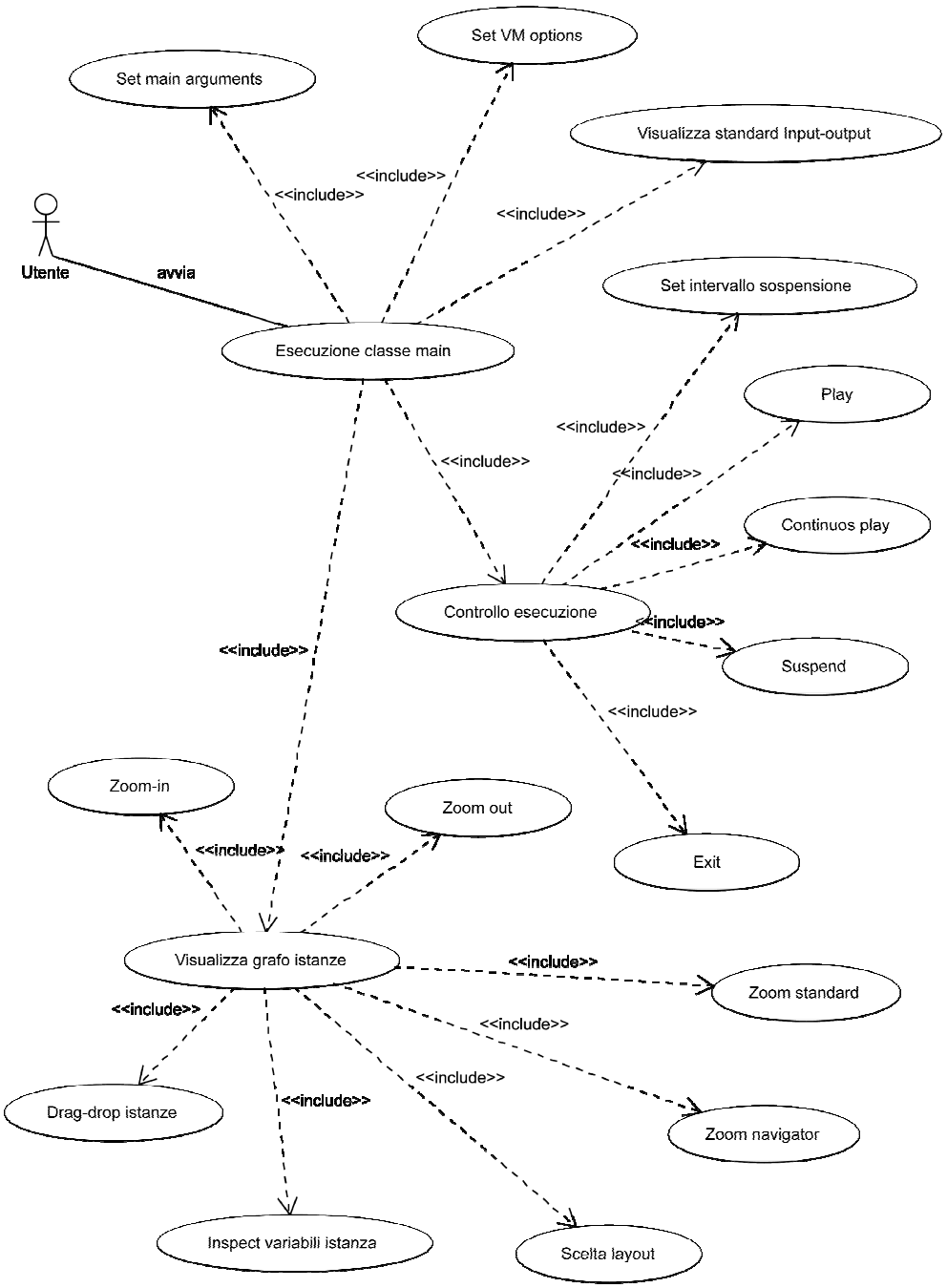
Analisi dei requisiti

Diagramma degli use-case

Analizzando i requisiti raccolti possiamo identificare un solo attore che corrisponde all'utente utilizzatore di AlgoExplorer. Possiamo quindi riportare i seguenti diagrammi degli use-case, in cui è stato utilizzato il termine "*workspace*" per indicare un nuovo insieme delle classi binarie e del relativo codice sorgente.



Use-Case: Configurazione Workspace



Use-Case: Esecuzione classe *main*

Diagramma degli stati e delle transizioni

Per quanto riguarda gli stati che può assumere l'esecuzione di un programma sotto il controllo di AlgoExplorer, si desume, attraverso i requisiti raccolti, il seguente diagramma degli stati e delle transizioni:

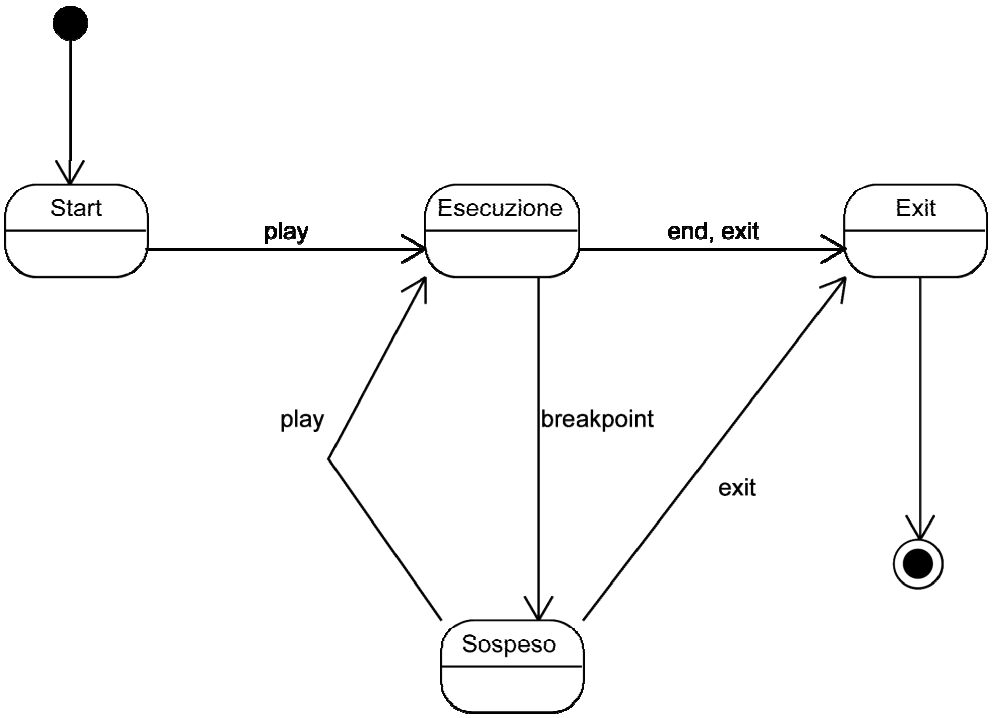


Diagramma degli stati e transizioni per l'esecuzione di un programma

Progettazione e sviluppo

La progettazione e lo sviluppo di AlgoExplorer sono stati suddivisi in due parti, così come logicamente suggerito dall'analisi dei requisiti degli use-case individuati. La prima parte è quella che si occupa della configurazione del workspace e quindi dell'interfaccia di caricamento delle classi, della visualizzazione del codice sorgente, dell'interfaccia per la scelta dei breakpoint e delle proprietà di visualizzazione delle classi scelte; la seconda parte è quella che si occupa di lanciare il programma che si vuole analizzare per studiarne le istanze e lo stato delle strutture dati di interesse, così come descritto nel capitolo dei requisiti funzionali.

Per entrambe queste parti è stato scelto di sviluppare l'interfaccia grafica utente utilizzando le API Java Swing, inoltre sono state utilizzate, come verrà spiegato più avanti, le API *"Java Debug Interface (JDI)"*[B4] di Sun per l'introspezione dei programmi esaminati da AlgoExplorer, le API *"Apache Byte Code Engineering Library"*[B5] per analizzare il bytecode delle classi binarie Java e le API *JGraph*[B6] per la visualizzazione grafica delle istanze degli oggetti; pertanto il disegno del diagramma delle classi fa riferimento a queste API.

Diagramma delle classi

Per praticità dividiamo il diagramma delle classi principali in due parti, descrivendone solo gli attributi e i metodi più importanti: la prima parte è quella che si occupa di tutta la logica di introspezione dei programmi in esecuzione da AlgoExplorer, la seconda si occupa del controllo dell'interfaccia grafica utente (GUI); pertanto si descrivono i seguenti due diagrammi.

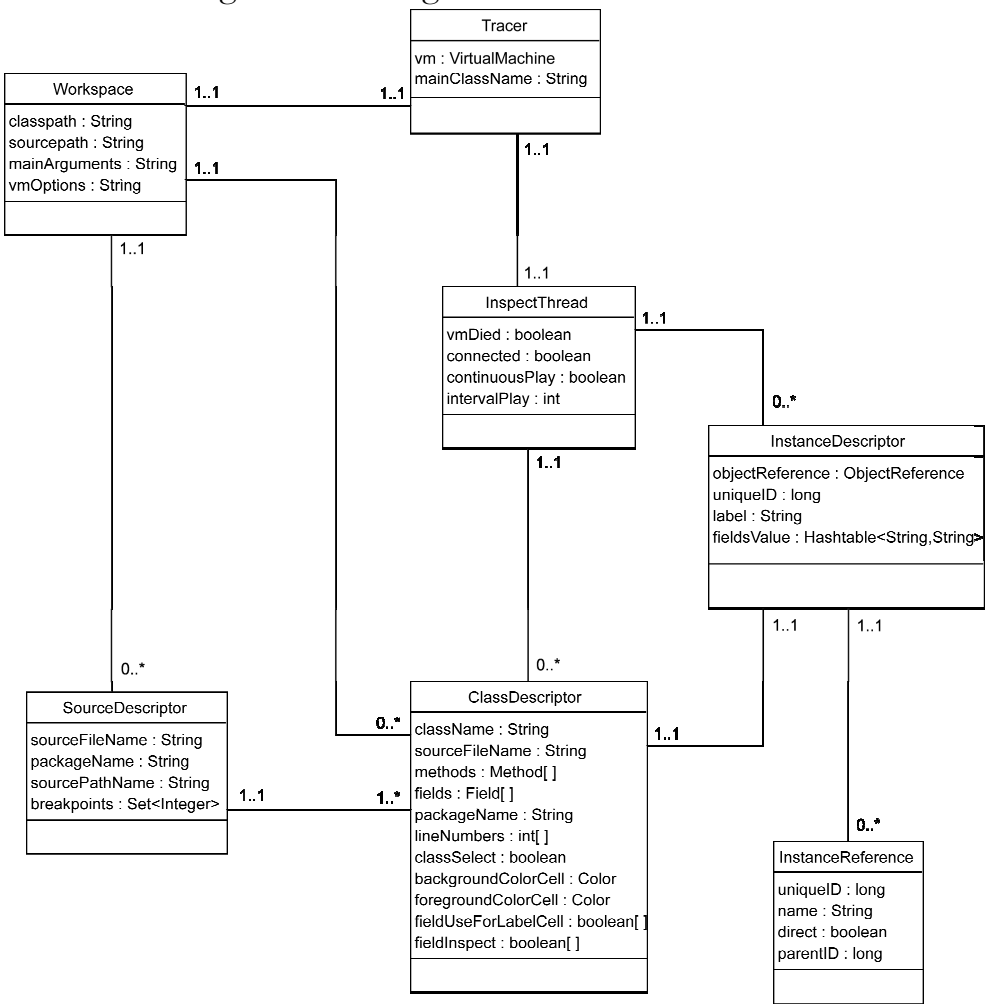


Diagramma delle Classi: logica di introspezione

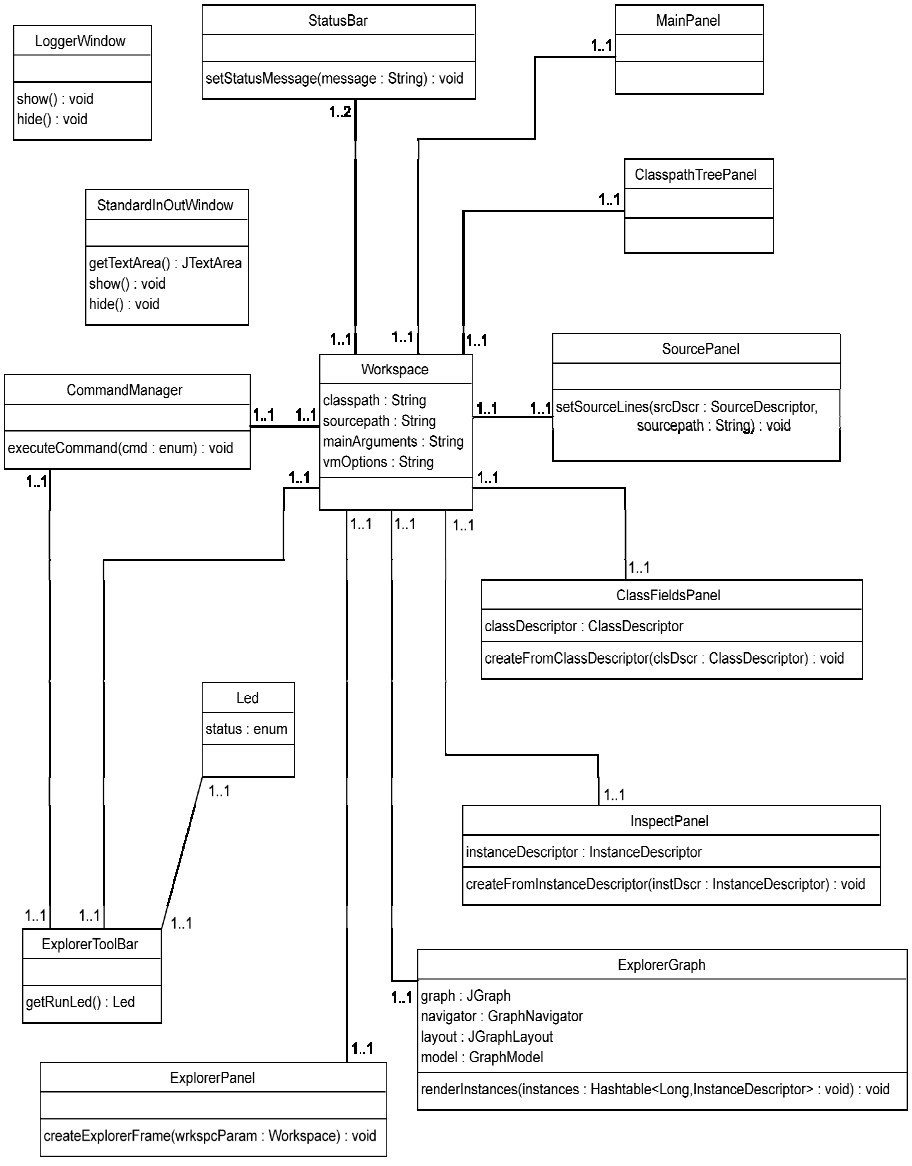
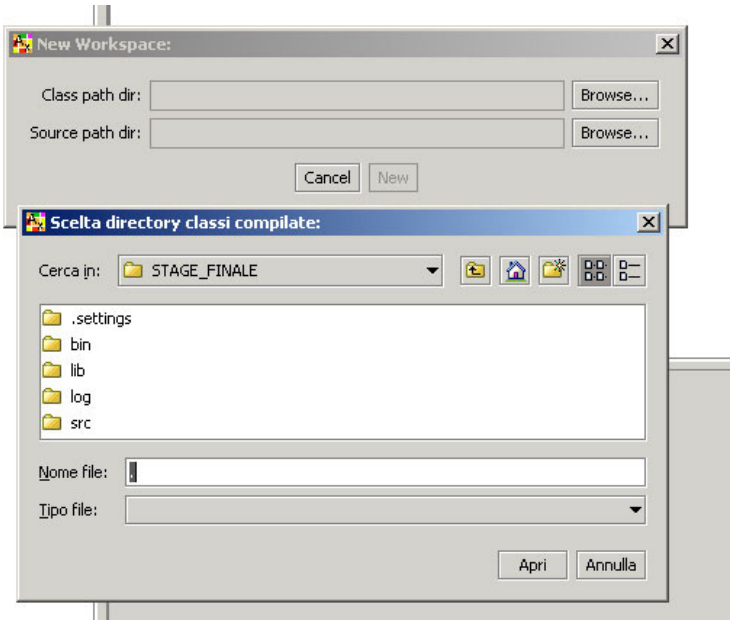


Diagramma delle Classi: interfaccia grafica utente (GUI)

Modulo configurazione workspace

Come descritto nei requisiti utente, una volta avviato AlgoExplorer, viene subito visualizzata una finestra che presenta l'interfaccia di configurazione del workspace. Il workspace è per AlgoExplorer un insieme di classi binarie e del rispettivo codice sorgente; pertanto la prima operazione che viene richiesta all'utente per iniziare a operare sugli algoritmi (o programmi) che vogliono essere analizzati, è il caricamento delle classi binarie Java e dei codici sorgenti. Attraverso la barra di menu è possibile accedere a questa inizializzazione: viene proposta la scelta delle cartelle contenenti il codice sorgente e il bytecode delle classi tramite una finestra di dialogo, come mostrato in figura, che utilizza il componente standard *javax.swing.JFileChooser*.



Inizializzazione di un nuovo workspace

L'inserimento della directory contente i file binari bytecode è obbligatoria, mentre l'inserimento della cartella dei file contenenti il relativo file sorgente è opzionale. AlgoExplorer permette di operare anche su programmi di cui non si dispone del relativo sorgente; in questo caso però non si potranno settare dei breakpoint sul codice sorgente e la visualizzazione delle strutture dati scelte avverrà solamente al termine del programma lanciato.

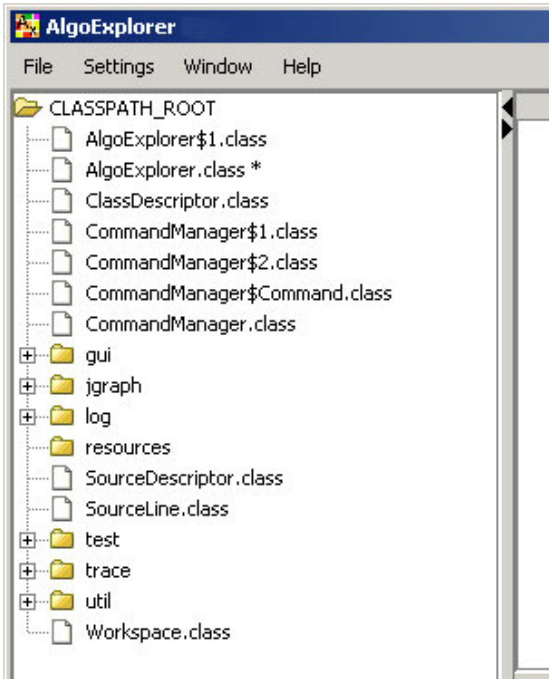
Le cartelle indicate a AlgoExplorer devono rispettare il percorso del classpath radice a cui fanno riferimento le classi importate; si dovrà quindi indicare la radice del classpath, mentre la directory dei file sorgenti dovrà rispettare la stessa struttura dei file bytecode. Questa struttura è quella normalmente utilizzata in molti IDE (Integrated Development Environment), come ad esempio in Eclipse[B7].

AlgoExplorer visita ricorsivamente e in profondità tutta la directory indicata per i file bytecode alla ricerca di classi binarie; questa ricerca non si limita semplicemente ad identificare una classe binaria quando si trova un file con l'estensione "**.class*", ma viene realmente verificato se il file in questione è conforme allo standard dei file Java bytecode. Inoltre i file bytecode trovati vengono analizzati per ricercare al loro interno il nome degli attributi sia pubblici che privati, il nome del file sorgente di riferimento e gli interi che rappresentano i numeri delle righe di codice effettivamente esecutivo del file sorgente da cui sono stati compilati. Per permettere di analizzare in questo modo un file

bytecode sono state utilizzate le API *"Apache Byte Code Engineering Library"*[B5] che permettono l'analisi, la creazione e la manipolazione dei file binari bytecode che rappresentano le classi compilate. E' stata utilizzata questa procedura, anziché le API Java Reflection (*java.lang.reflect*)[B8], per diversi motivi: in primo luogo perché, per utilizzare la Java Reflection, bisogna leggere la classe che si vuole esaminare utilizzando una chiamata al metodo statico *Class.forName*; in questo modo, la classe viene caricata all'interno della Java Virtual Machine (JVM) in esecuzione, che legge il bytecode e lo rende disponibile all'esecuzione così come per qualsiasi altra classe utilizzata dalla JVM, facendo si che ogni dipendenza o riferimento esterno della classe sia a sua volta caricato. Quindi, utilizzando questo metodo, se si analizzasse una classe che ha un riferimento ad una classe non presente nel classpath di AlgoExplorer, si incorrerebbe in un'eccezione di tipo *ClassNotFoundException*. Questo effetto collaterale non è generato dalle API Apache Byte Code Engineering Library che invece si limitano a leggere il bytecode di una classe come fosse un qualsiasi file, senza ricercarne le dipendenze. In secondo luogo la Java Reflection non è stata utilizzata perché con questa tecnica non si possono conoscere i nomi degli attributi o dei metodi della classe che sono privati, capacità che invece hanno le API Apache Byte Code Engineering Library; inoltre questa libreria è capace di restituire i numeri delle linee di codice, che rappresentano le

istruzioni esecutive del codice sorgente da cui è stata compilata la classe, e il nome del file sorgente stesso: questa capacità della libreria è applicabile comunque su qualsiasi classe compilata a meno dell'opzione standard "*-g:none*" del comando di compilazione *javac*.
La possibilità di conoscere i nomi degli attributi privati è di notevole interesse per l'applicazione AlgoExplorer, in quanto è desiderabile poter analizzare anche questi riferimenti. Le API Apache Byte Code Engineering Library, oltre a poter analizzare come è realizzata una classe, sono capaci anche di modificare il bytecode delle classi, variandone i metodi o gli attributi, ad esempio aggiungendo delle chiamate ad altri metodi esterni, senza avere a disposizione il codice sorgente della classe che si modifica.
Come già descritto, quindi, AlgoExplorer ricerca all'interno della directory indicata i file delle classi compilate e ne memorizza in istanze di oggetti della classe *ClassDescriptor* (vedi diagramma delle classi) i nomi degli attributi e il tipo, mentre il nome del file sorgente dal quale la classe è stata compilata e i numeri delle righe esecutive di questo file sono memorizzati in istanze di oggetti della classe *SourceDescriptor*; AlgoExplorer utilizza queste informazioni per permettere di scegliere le variabili che si vogliono ispezionare in fase di esecuzione.
Una volta visitata la directory dei file bytecode delle classi, viene presentata una struttura ad albero con cui è possibile esplorare i

package contenuti nel classpath, come mostrato nella seguente figura.



Albero dei package e delle classi

Le classi che contengono un metodo *main*, e quindi eseguibili, vengono contrassegnate con un asterisco.

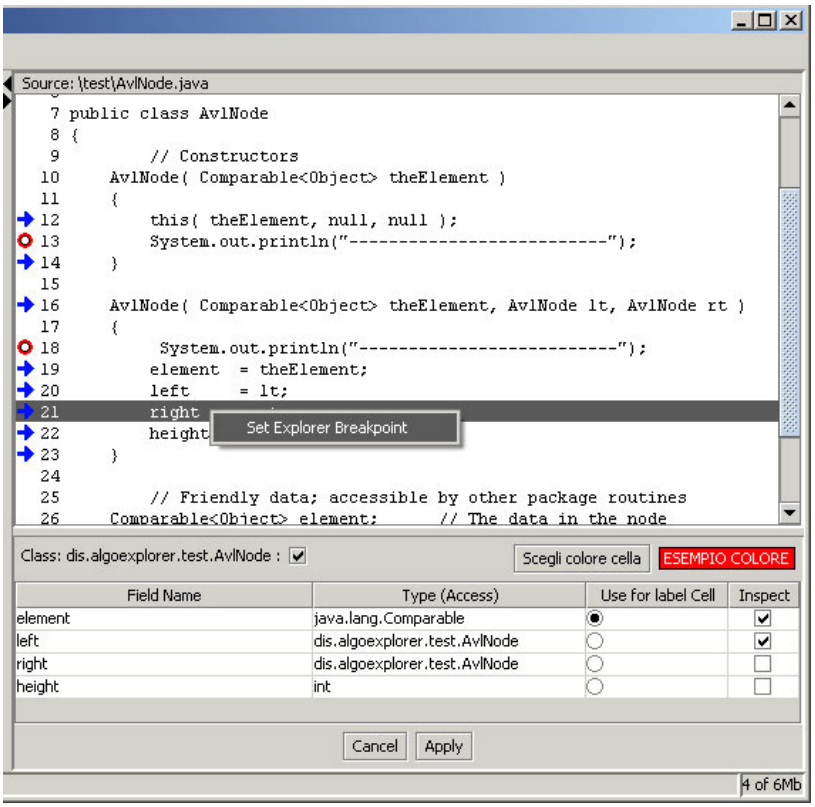
Cliccando sul nome di una classe, viene caricato in un pannello laterale il codice sorgente della classe, se disponibile. Inoltre vengono presentate, in un altro pannello, le impostazioni di visualizzazione delle istanze della classe. Come descritto nei requisiti utente, è possibile scegliere di visualizzare le istanze della classe, il colore di visualizzazione e l'attributo da utilizzare, tra quelli della classe stessa, per l'etichetta dell'istanza; qualora non si scelga

nessun attributo, l'etichetta sarà generata utilizzando il metodo *toString* delle istanze della classe. In questo pannello viene poi data la possibilità di scegliere quali tra gli attributi della classe saranno ispezionabili. Oltre a queste impostazioni, tramite un menu contestuale sulle righe del codice presentato, visualizzabile cliccando il tasto destro del mouse su una riga, è possibile andare a inserire o eliminare i breakpoint di interesse nel codice sorgente; è possibile inserire i breakpoint solamente nelle righe di codice che presentano sulla sinistra della riga stessa, un simbolo grafico rappresentante una piccola freccia di colore blu; le righe contrassegnate con questo simbolo sono le sole righe eseguibili del codice sorgente e sono righe individuate tramite le API Apache Byte Code Engineering Library, utilizzando il metodo *getLineNumberTable()* della classe *org.apache.bcel.classfile.LineNumberTable*. Accedendo al menu contestuale per impostare un breakpoint su una riga eseguibile scelta, verrà visualizzato, sempre sulla sinistra della riga un simbolo rappresentante un piccolo cerchio rosso, per indicare che quella riga è una riga in cui è stato settato un breakpoint; solamente all'esecuzione delle righe indicate come breakpoint, AlgoExplorer visualizzerà le differenze dello stato delle istanze delle classi scelte.

Premendo il pulsante di conferma, tutte le impostazioni che l'utente ha effettuato vengono salvate sulle istanze delle classi *ClassDescriptor* e *SourceDescriptor*, in modo da poter essere utilizzate successivamente

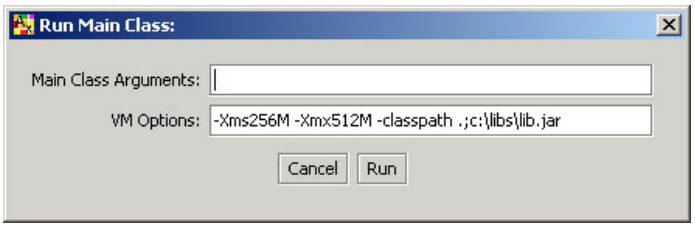
nella fase di esecuzioni del programma scelto; i breakpoint invece vengono automaticamente salvati nel momento stesso in cui vengono impostati.

Nella figura seguente viene visualizzata la finestra con il pannello che presenta il codice sorgente e l'interfaccia di scelta dei parametri di visualizzazione delle istanze della classe.



Interfaccia di impostazione delle classi visualizzabili

Una volta effettuate le scelte di visualizzazione, tramite un menu contestuale sull'albero dei package e delle classi, cliccando con il tasto destro del mouse una classe contrassegnata come classe contenente il metodo *main*, è possibile, selezionando il comando "Run", iniziare l'esecuzione sotto il controllo di AlgoExplorer. Tramite una finestra di dialogo vengono visualizzate le impostazioni di esecuzione del programma scelto: si possono specificare eventuali argomenti da passare al metodo main del programma e eventuali opzioni della JVM, come ad esempio impostazioni sulla configurazione della memoria allocata per lo Heap o l'impostazione di librerie esterne utilizzate nell'esecuzione dal programma tramite l'opzione "classpath".



Impostazioni di esecuzione di una classe *main*

Una volta effettuate le scelte desiderate, premendo il pulsante "Run", AlgoExplorer eseguirà il programma nella modalità di introspezione delle strutture dati che si sono selezionate.

Scelta della modalità di introspezione dei programmi

Per permettere ad AlgoExplorer di poter esaminare le istanze delle classi che si sono scelte e la loro conseguente visualizzazione, si sono esaminate le seguenti modalità:

1. Bytecode Instrumentation (BCI) [B10]
2. Java Virtual Machine Tool Interface (JVMTI) [B9]
3. Java Debug Interface (JDI) [B4]

Per ognuna di queste modalità si riporta di seguito una breve descrizione e uno studio sui vantaggi e/o svantaggi.

Bytecode Instrumentation (BCI):

BCI[B10] prevede la manipolazione, attraverso opportune librerie (API *"Apache Byte Code Engineering Library"*[B5]), del bytecode delle classi compilate, o immediatamente dopo la compilazione o in fase di esecuzione.

Per riuscire ad evidenziare le azioni dell'algoritmo che si vorrà analizzare, è necessario scrivere le classi che rappresentano le classi delle istanze visualizzabili secondo le specifiche JavaBean: attributi privati, metodi *get/is* e *set* per accedere agli attributi.

In questo modo però si costringe lo sviluppatore ad adottare questa tecnica di programmazione per rappresentare le strutture dati che si vogliono visualizzare, il che non rientra nelle specifiche richieste.

Esaminando questa procedura si evidenziano i seguenti punti:

- Facilità di realizzazione.
- Eventuale utilizzo di Dynamic Bytecode Instrumentation (DBI)[A3], con il quale è possibile effettuare la procedura Bytecode Instrumentation dinamicamente in fase di esecuzione della JVM (es: utilizzando l'opzione "java -avaagent:instrument.jar").
- Scarsa flessibilità, in quanto ogni classe delle strutture dati deve obbligatoriamente essere un JavaBean.
- Impossibilità nell' identificazione di array e di conoscere tutte le istanze che referenziano un certo oggetto.

Java Virtual Machine Tool Interface (JVMTI):

JVMTI[B9] è un'interfaccia nativa che permette l'analisi e il controllo dell'esecuzione delle applicazioni sulla JVM; supporta il controllo completo degli stati della JVM comprendendo funzioni di profiling, debugging, monitoring e analisi/controllo dei thread. E' parte della Java Platform Debugger Architecture (JPDA)[B11] introdotta a partire dalla versione di Java 1.5.

Esaminando questa interfaccia si evidenziano i seguenti punti:

- Permette l'identificazione di ogni evento, come la creazione di un oggetto, la sua modifica, la creazione di array, ecc.
- E' adatta per lo studio del debugging e profiling di tutto il comportamento della JVM.

- E' nativa e pertanto il suo utilizzo richiede un codice scritto in C/C++.
- Utilizzo e sviluppo sono decisamente complessi.
- Viene sconsigliata dalla stessa Sun per il debug sul codice applicativo, per il quale viene consigliata o la procedura di Dynamic Bytecode Instrumentation (DBI) o l'utilizzo delle librerie Java Debug Interface (JDI).

Java Debug Interface (JDI):

JDI[B4] è una libreria (anche questa parte di JPDA[B11]) completamente in Java che permette l'introspezione dello stato della JVM, delle classi, degli array, delle interfacce, dei tipi primitivi e delle istanze di questi tipi; provvede inoltre al controllo esplicito dell'esecuzione della JVM, al suspend e al resume di thread, all'arresto in presenza di brakpoint, alla notifica delle eccezioni, del caricamento di classi e della creazione dei thread. Tutte le variabili applicative (istanze di oggetti, tipi primitivi, array) sono ispezionabili attraverso questa interfaccia, che assegna automaticamente ad ogni istanza di una classe un id univoco; inoltre è possibile conoscere anche le istanze che direttamente referenziano l'istanza in esame: in questo modo si può sapere se le istanze sono contenute in array, Vector, List, Hashtable, ecc., e da quale altra istanza applicativa viene referenziata.

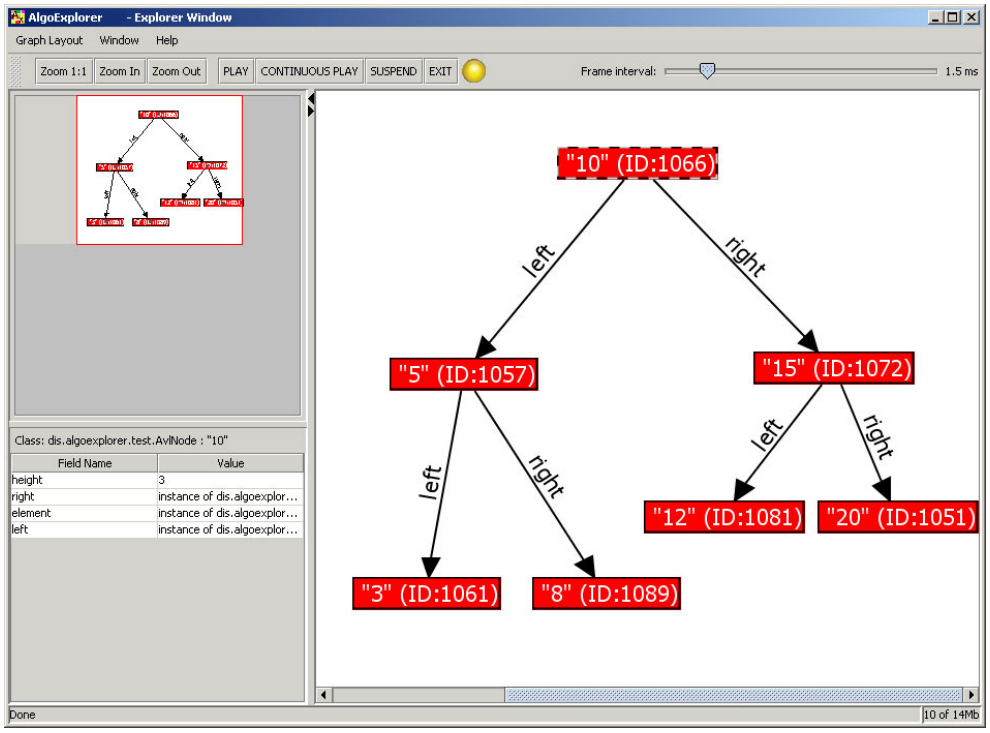
Esaminando questa interfaccia si evidenziano i seguenti punti:

- Facilità di utilizzo, anche se poco documentata da SUN (esiste infatti solo solo la Javadoc delle API e un esempio di codice applicativo).
- Portabilità completa su qualsiasi piattaforma (è presente su ogni JRE rilasciata da SUN e ogni JVM, rilasciata da altri vendor, deve implementarla).
- Permette l'introspezione di qualsiasi variabile, anche array.
- Identifica ogni istanza con un id univoco e restituisce in modo semplice gli oggetti che referenziano un'istanza data.

Da quanto analizzato, l'interfaccia JDI è la più adatta per i nostri scopi, in quanto permette un'ampia introspezione di tutte quelle che potrebbero essere le variabili e le istanze rappresentabili negli algoritmi che si vogliono visualizzare tramite AlgoExplorer.

Modulo introspezione dei programmi

Così come descritto nei requisiti utente, all'avvenuto lancio di una classe che contiene il metodo *main*, viene creata una nuova finestra che rappresenta l'ambiente grafico di esecuzione, visualizzazione e controllo del programma scelto che si vuole andare a analizzare. La seguente figura mostra come si presenta questa GUI, chiamata "*Explorer Window*", durante l'esecuzione di un algoritmo che opera su alberi AVL.



Finestra di esecuzione e controllo dei programmi

Come descritto in precedenza, per analizzare l'esecuzione di un programma si è scelto di utilizzare l'interfaccia JDI che permette il controllo completo e l'introspezione di tutte le strutture dati manipolate dal programma in esame. In AlgoExplorer le classi che si occupano del controllo dell'esecuzione dei programmi sono la classe *Tracer* e la classe *InspectThread*.

La classe *Tracer* si occupa di inizializzare l'interfaccia JDI e preparare un'istanza della classe *com.sun.jdi.VirtualMachine* che si occuperà di creare una nuova JVM, sotto il controllo di JDI, che a sua volta eseguirà il programma richiesto dall'utente. In questa fase di inizializzazione, vengono assegnati alla nuova JVM sia gli

argomenti scelti per il metodo *main* della classe che andrà in esecuzione, sia le opzioni scelte della JVM, come descritto nella sezione precedente riguardo le impostazioni di esecuzione di una classe. La classe *Tracer* inoltre si occupa di inizializzare anche un'istanza della classe *InspectThread*, vero cuore della logica di introspezione dei programmi di AlgoExplorer. Una volta fatte tutte le inizializzazioni, la classe *Tracer* esegue, attraverso il metodo *start*, l'istanza del thread *InspectThread*.

JDI utilizza un'architettura ad eventi, pertanto il costruttore della classe *InspectThread* inizializza i listener sugli eventi che si vogliono andare ad osservare; in primo luogo viene inizializzato un listener sull'evento *com.sun.jdi.event.MethodExitEvent* e sull'evento *com.sun.jdi.event.ClassPrepareEvent*. Per riuscire ad osservare, attraverso l'evento *MethodExitEvent*, tutte le istanze delle classi scelte per essere visualizzate ad ogni *exit* sui metodi costruttori di una classe, si controlla se quella classe rientra tra le classi che si sono scelte per la visualizzazione; nel caso lo sia, allora viene creata un'istanza della classe *InstanceDescriptor* (come da diagramma delle classi riportato precedentemente) che viene poi memorizzata all'interno di una un attributo di tipo *Hashtable* di nome "*instances*".

La classe *InspectThread*, una volta effettuata l'inizializzazione sui listener descritti precedentemente, esegue un ciclo su una coda di eventi gestita dall'interfaccia JDI; viene atteso sulla coda un nuovo evento da esaminare e reindirizzato al corrispettivo metodo che

dovrà gestirlo tramite il metodo *handleEvent* della classe *InspectThread*. E' possibile stabilire se l'esecuzione della JVM sotto il controllo JDI debba essere sospesa ogni volta che viene lanciato un evento per cui si è creato un listener, oppure no; nel caso di AlgoExplorer si è scelto di gestire tutti gli eventi in modalità sospesa, in modo da non incorrere in possibili eventi che si andrebbero a sovrapporre, se questa non è stata ancora terminata, con l'esecuzione dei metodi di AlgoExplorer che esaminano gli eventi lanciati; inoltre, utilizzando questa modalità, è possibile gestire, tramite i pulsanti play, continuous play e suspend, il controllo dell'esecuzione del programma che si vuole esaminare così come richiesto nelle specifiche.

L'evento *ClassPrepareEvent*, invece, viene lanciato ogni volta che la JVM che esegue il programma incontra un'istanza di un tipo di classe che non aveva ancora inizializzato; l'azione associata a questo evento è quella di settare i listener sui breakpoint inseriti nel codice (*BreakpointEvent*) e sugli attributi che si vogliono andare ad ispezionare durante l'esecuzione di un programma (*ModificationWatchpointEvent*).

Quando incontrata un breakpoint, JDI lancia l'evento *BreakpointEvent* e AlgoExplorer esegue il metodo *updateInstancesReferencesAndLabel*, che si occupa di andare ad analizzare tutte le istanze di tipo *InstanceDescriptor* memorizzate nell'*Hashtable instances*; per ogni istanza, chiamando il metodo

toString su l'attributo che si è scelto come riferimento, viene aggiornata la label che dovrà essere visualizzata e vengono anche aggiornate tutte le referenze dell'istanza in esame tramite il metodo ricorsivo *findReferences*. Il metodo *findReferences* visita ricorsivamente, fino ad un numero di livelli preimpostato e costante, cercando tutti gli oggetti che referenziano l'istanza data attraverso il metodo JDI "*referringObjects*" della classe *ObjectReference*: ogni istanza trovata viene memorizzata in una nuova istanza della classe *InstanceReference* e inserita in una *HashSet* gestita dalla classe *InstanceDescriptor*.

Una volta che il metodo *updateInstancesReferencesAndLabel* ha aggiornato tutte le istanze memorizzate nell'attributo "*instances*", viene eseguito il metodo *renderInstances* della classe *ExplorerGraph*, che si occupa della visualizzazione delle istanze e dei collegamenti tra queste. Si è scelto di usare le API JGraph[B6] per la visualizzazione grafica delle istanze degli oggetti. Questa libreria è capace di gestire la visualizzazione dei grafi complessi, raffigurando il posizionamento automatico dei nodi con diversi tipi di layout (es: come albero o come grafo). Il metodo *renderInstances* controlla le modifiche effettuate nelle istanze di tipo *InstanceDescriptor* e *InstanceReference* ed effettua gli eventuali aggiornamenti e la visualizzazione sui nodi del grafo tramite le API JGraph. Il metodo *renderInstaces*, oltre a visualizzare le modifiche delle istanze, interroga, per ogni *ObjectReference* di queste, il metodo *isCollected* per verificare se l'istanza in esame è passata sotto il controllo della Garbage

Collection; in caso affermativo, l'istanza viene visualizzata concatenando un asterisco alla fine della label che la rappresenta. E' tuttavia possibile disabilitare la Garbage Collection tramite il file di impostazioni di AlgoExplorer, descritto nella sezione seguente.

Classi di supporto

E' stata creata una classe, chiamata *ConstantManager*, che si occupa della gestione di tutte le costanti di configurazione di AlgoExplorer. Viene letto, all'inizializzazione di questa classe, un file di properties, chiamato *"algoexplorer.properties"*, contenente tutte le costanti di interesse dell'applicazione; le costanti vengono poi memorizzate in attributi di tipo public static final accessibili così a qualunque classe di AlgoExplorer.

Tutti i log dell'applicazione sono gestiti dalle API Apache log4j[B12] e reindirizzati, attraverso la classe *TextAreaAppender*, in una textarea visibile tramite una finestra adibita proprio alla visualizzazione del log applicativo di AlgoExplorer e richiamabile con la barra del menu; in questo modo, tramite il file di configurazione *log4j.properties* di Log4j, si può impostare il livello di verbose del log applicativo.

Analisi delle prestazioni

Per analizzare le prestazioni di AlgoExplorer si è utilizzato il tool di profiling Java VisualVM[B13]. Non si è riscontra alcuna

problematica prestazionale dovuta alle scelte progettuali effettuate, e l'impiego delle risorse utilizzate da AlgoExplorer rimane direttamente proporzionale alle risorse richieste dai programmi che si vogliono analizzare.

Si deve tuttavia evidenziare che il listener, sviluppato sull'evento *com.sun.jdi.event.MethodExitEvent*, per effettuare il controllo delle istanze inizializzate, è chiamato su tutti i metodi delle classi dell'applicazione analizzata; questa procedura pertanto rallenta i programmi in esecuzione controllati da AlgoExplorer; infatti le applicazioni sotto il controllo di JDI risultano lente se sottoposte a frequenti eventi gestiti da JDI.

AlgoExplorer è stato interamente sviluppato utilizzando l'IDE Eclipse[B7] versione 3.5 e testato su piattaforma Windows XP SP3 e Sun JDK versione 1.6.0_07.

Descrizione dettagliata dell'interfaccia utente realizzata

Appena avviato AlgoExplorer presenta la finestra principale che permette la scelta delle variabili e delle classi da analizzare. Tramite una barra di menu si può accedere alla funzione di importazione delle classi binarie e del codice sorgente: una finestra di dialogo chiede di indicare le directory in cui sono presenti i file sorgenti java e le classi compilate. Una volta specificati questi due percorsi AlgoExplorer visita la directory delle classi compilate, visualizzando in una struttura ad albero i package trovati e le relative classi

contenute. Tutte le classi che contengono un metodo *main* sono evidenziate in modo diverso all'interno dell'albero. Selezionando una classe tra quelle visualizzate nell'albero è possibile visualizzare il codice sorgente (se è stato trovato il relativo file Java sorgente) e i nomi di tutti gli attributi della classe stessa. Nella finestra che visualizza il codice sorgente, tramite un menu contestuale accessibile cliccando il tasto destro del mouse sulle righe del sorgente, è possibile indicare gli eventuali punti di breakpoint in cui si vuole che AlgoExplorer visualizzi lo stato degli oggetti e delle variabili; un apposito indicatore sulla linea scelta evidenzia che la linea rappresenta un breakpoint. I breakpoint inseriti si possono rimuovere utilizzando il menu contestuale sulla riga del codice sorgente in cui sono presenti.

Nel caso non sia stato inserito nessun breakpoint all'interno del codice sorgente, la visualizzazione dello stato delle strutture dati scelte avverrà solamente al termine del metodo *main* del programma lanciato.

In una finestra separa dalla quella di visualizzazione del codice sorgente viene visualizzata una tabella con i nomi di tutti gli attributi della classe, con i vari tipi di accesso (es: pubblico o privato) e con il tipo a cui appartengono le relative variabili. Tramite un apposito checkbox è possibile specificare se si ha interesse che le istanze della classe selezionata siano visualizzate, e, in questo caso, è anche possibile scegliere i colori che verrano utilizzati per lo sfondo

e per il testo dell'istanza. Nella stessa finestra è possibile scegliere uno degli attribuiti come campo dell'etichetta che verrà visualizzata per le istanze delle classi; in questo caso l'etichetta conterrà il valore restituito dal metodo *toString* chiamato sull'attributo scelto. Nel caso non venisse scelto alcun attributo, il metodo *toString* sarà direttamente chiamato sull'istanza della classe. Inoltre è possibile selezionare uno o più attributi che si potranno ispezionare per conoscerne il valore nei diversi breakpoint individuati sul codice sorgente. Una volta effettuate le scelte di visualizzazione, tramite un apposito pulsante è possibile salvare le modifiche. Nel caso si volessero eliminare le scelte di visualizzazione per una certa classe, cliccando sull'albero delle classi con il tasto destro del mouse, si ha accesso ad un menu contestuale che permette di rimuovere le impostazioni eventualmente salvate.

Una volta effettuate le scelte di visualizzazione desiderate si può passare all'esecuzione di una delle classi visibili nell'albero che identifica i packages e le classi al suo interno. La classe che si vuole eseguire deve contenere il metodo *main* per essere eseguita, pertanto solo le classi evidenziate possono essere lanciate; cliccando con il tasto destro del mouse un menu contestuale permette di eseguire la classe scelta. Una finestra di dialogo chiede di specificare eventuali parametri da passare al metodo *main*, che si sta per eseguire, e di indicare eventuali opzioni per la Java Virtual Machine che eseguirà il programma: è possibile indicare, ad esempio, un classpath per

specificare librerie jar e/o indicare opzioni proprio della Java Virtual Machine che si sta utilizzando; la sintassi di queste opzioni è la stessa di quella che viene normalmente utilizzata al lancio di una classe Java da linea di comando. Una volta inseriti i valori desiderati, tramite un pulsante su quest'ultima finestra di dialogo, si avvierà la modalità di esecuzione dell'algoritmo e di visualizzazione di AlgoExplorer.

Nella modalità di esecuzione di AlgoExplorer, viene aperta un'ulteriore finestra, in cui è presente una barra di comandi tramite la quale è possibile controllare l'esecuzione del programma scelto e lo zoom sull'area di visualizzazione. Più precisamente, si ha un pulsante per lo zoom-in sull'area di lavoro, un pulsante per lo zoom-out e un pulsante per ripristinare la visualizzazione nella scala di default. Accanto ai pulsanti sono presenti i comandi per il controllo dell'esecuzione; un programma può essere in stato di esecuzione, di sospensione o terminato. Sull'interfaccia è presente una segnale colorato che indica lo stato in cui il programma si trova. Un segnale verde continuo indica che il programma è appena stato avviato ma è ancora in stato di sospensione; un segnale giallo continuo indica che il programma è in esecuzione; un segnale lampeggiante tra verde e giallo indica che il programma è in stato di sospensione a seguito di un breakpoint impostato sul codice sergente; un segnale rosso continuo indica che il programma è terminato.

Inizialmente un programma è in sospensione e solo premendo il pulsante "play" si avvia l'esecuzione. Ad ogni breakpoint impostato sul codice sorgente, l'esecuzione si arresta per visualizzare lo stato delle istanze delle classi selezionate per la visualizzazione. Sull'interfaccia sono inoltre presenti: un pulsante "continuous play" che avvia l'esecuzione nel caso sia in stato sospesa, visualizza lo stato delle istanze nei breakpoint sul codice sorgente e sospende l'esecuzione solo per un numero di decimi di secondo impostato tramite una barra slider sulla barra dei comandi; un pulsante "suspend" che permette di sospendere al primo breakpoint incontrato il tipo di esecuzione "continua" impostata dal pulsante "continuous play"; un pulsante "exit" che arresta e termina l'esecuzione del programma.

In ogni stato di esecuzione del programma è possibile cliccare le istanze visualizzate e agire sui pulsanti di zoom o sul pannello di navigazione; quest'ultimo è un pannello che visualizza in scala molto ridotta l'intera area di visualizzazione e che permette di impostare l'area scelta semplicemente cliccando e trascinando un rettangolo che rappresenta l'area visibile.

Cliccando sui rettangoli che rappresentano le istanze delle classi scelte per la visualizzazione, vengono caricati su un pannello laterale i valori degli attributi selezionati per quella classe tra quelli scelti tra i possibili ispezionabili.

Tramite un menu di selezione si può scegliere tra diversi tipi di layout automatico delle istanze, ad esempio come albero o come grafo, o impostare una modalità manuale. In ogni istante è possibile cliccare un rettangolo che rappresenta un'istanza e posizionarlo dove si desidera semplicemente trascinandolo.

I rettangoli che rappresentano le istanze delle classi scelte per la visualizzazione, sono descritti con un etichetta che contiene la stringa nella maniera scelta in fase di impostazione, come precedentemente descritto; inoltre è presente un identificatore univoco che rappresenta l'istanza della classe. I rami che collegano le istanze delle classi rappresentano, invece, le referenze di un'istanza ad un'altra; anche in questo caso sono presenti delle etichette che rappresentano il nome della variabile che contiene la referenza all'istanza puntata. Nel caso l'istanza puntata non sia direttamente referenziata dalla variabile di un oggetto, ma sia referenziata in modo *"indiretto"* (es: una classe contiene un attributo di tipo java.util.List<E> che a sua volta contiene una referenza ad un oggetto), allora la linea che rappresenta il ramo è visualizzata in modo tratteggiato, altrimenti il ramo è visualizzato in modo continuo (referenza *"diretta"*).

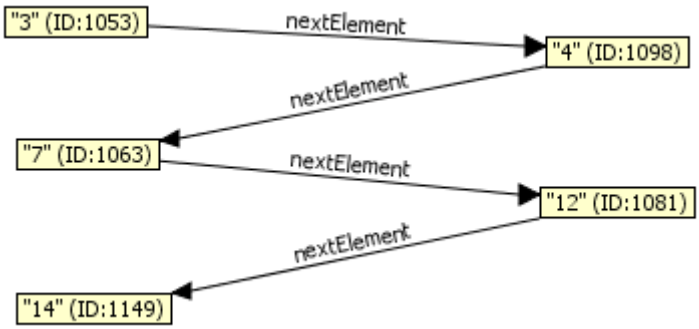
In fase di esecuzione di un programma lanciato con AlgoExplorer, viene aperta una finestra che ridireziona lo standard input-output testuale del programma.

Esempi d'uso su algoritmi conosciuti

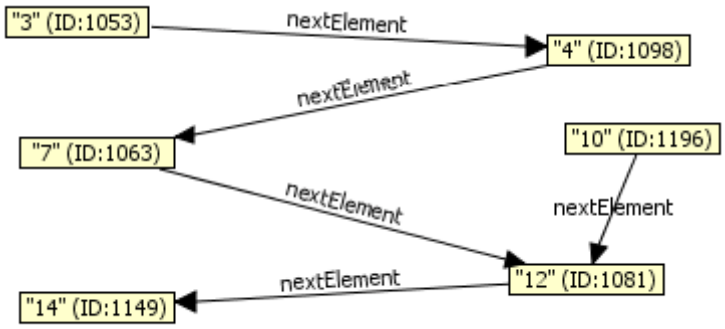
In questo capitolo si descrive l'uso di AlgoExplorer a scopo didattico, su algoritmi ben conosciuti, per evidenziare il possibile impiego per la comprensione interattiva dei programmi.

Lista Ordinata

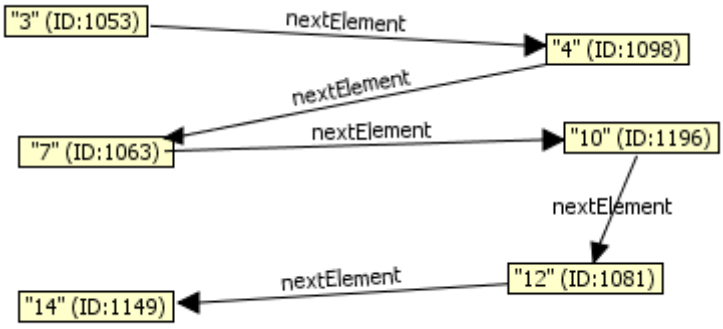
Si è analizzato l'algoritmo *OrderedList*, desunto dal codice sorgente in [A4] e [B14], che opera l'inserimento di elementi in una lista ordinata. L'algoritmo, all'inserimento di un elemento, visita la lista fino a trovare un elemento maggiore dell'elemento che si sta inserendo, in modo che quest'ultimo venga inserito immediatamente prima dell'elemento trovato. E' stato implementato un input del numero degli elementi da inserire e un input degli elementi stessi tramite un *JOptionPane*, in modo da poter permettere di interagire con i valori che si vogliono inserire. Vengono riportate di seguito alcune schermate dell'area di visualizzazione di AlgoExplorer, al fine di evidenziare i passi eseguiti dall'algoritmo.



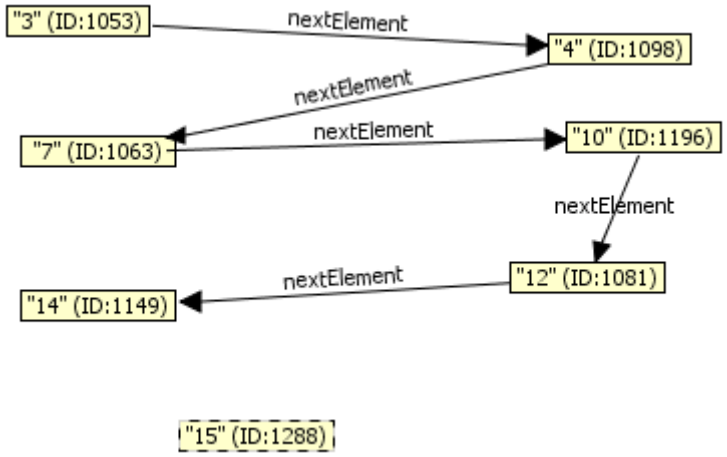
Lista contente 5 elementi pre-inseriti



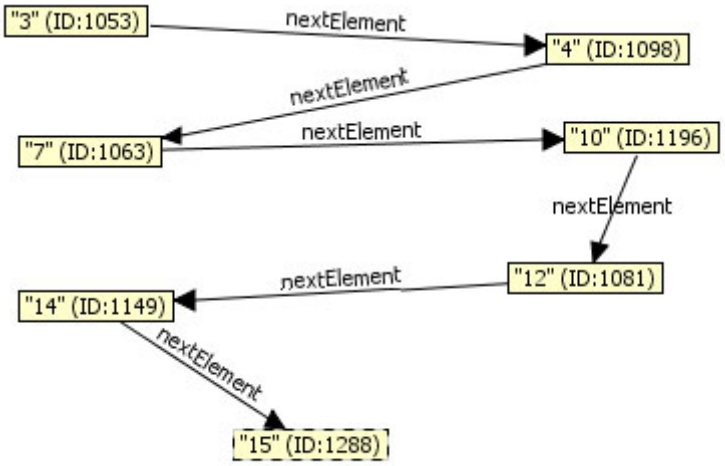
Creazione dell'elemento con valore 10 e inizio inserimento



Completamento dell'inserimento dell'elemento con valore 10



Creazione dell'elemento con valore 15

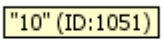


Completamento dell'inserimento in coda dell'elemento con valore 15

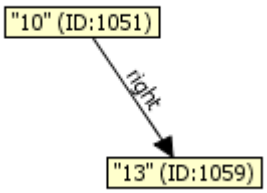
Albero bilanciato AVL

Si è analizzato l'algoritmo di inserimento dei nodi su un albero bilanciato di tipo AVL, desunto dal codice sorgente in [A5] e [B15]; anche in questo caso, tramite un input gestito da un *JOptionPane*, è possibile interagire con i valori inseriti. Sono stati studiati due casi: il primo su un inserimento di tipo Right Right Case (DD) e il secondo su un inserimento di tipo Left Right Case (DS); di seguito vengono riportate le schermate della visualizzazione di AlgoExplorer.

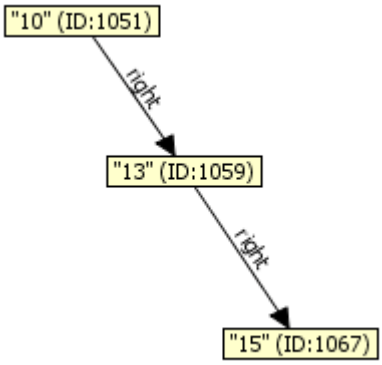
Inserimento Right Right Case (DD):



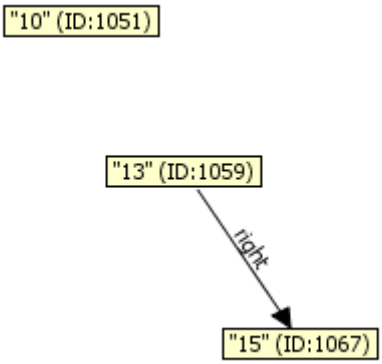
Inserimento nodo 10



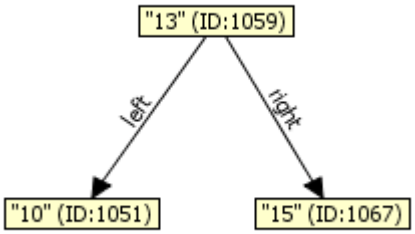
Inserimento a destra di un nodo 13



Inizio inserimento nodo 15

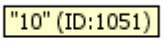


Inizio rotazione sinistra

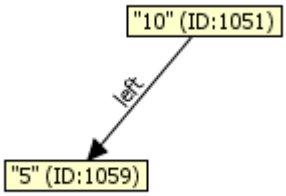


Completamento dell'inserimento con rotazione sinistra

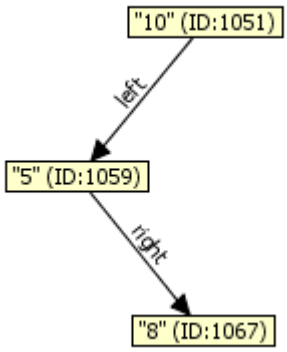
Inserimento Left Right Case (DS):



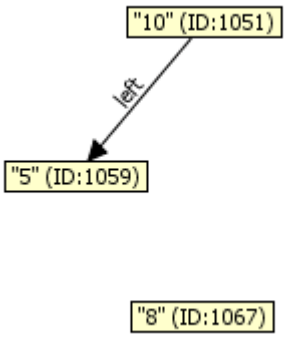
Inserimento nodo 10



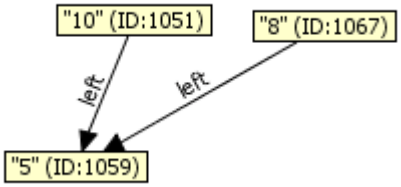
Inserimento a sinistra di un nodo 5



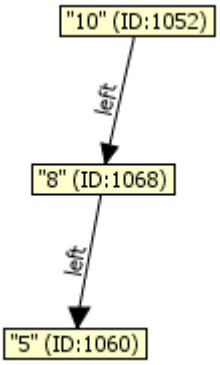
Inizio inserimento nodo 8



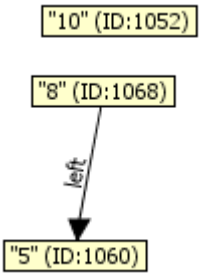
Inizio rotazione sinistra



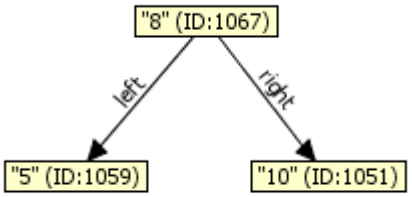
Fase intermedia rotazione sinistra



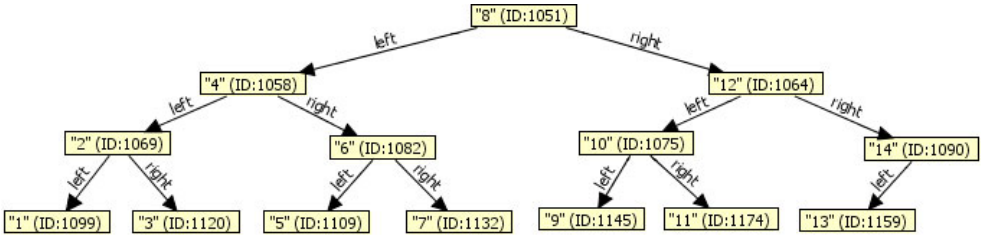
Completamento rotazione sinistra



Inizio rotazione destra



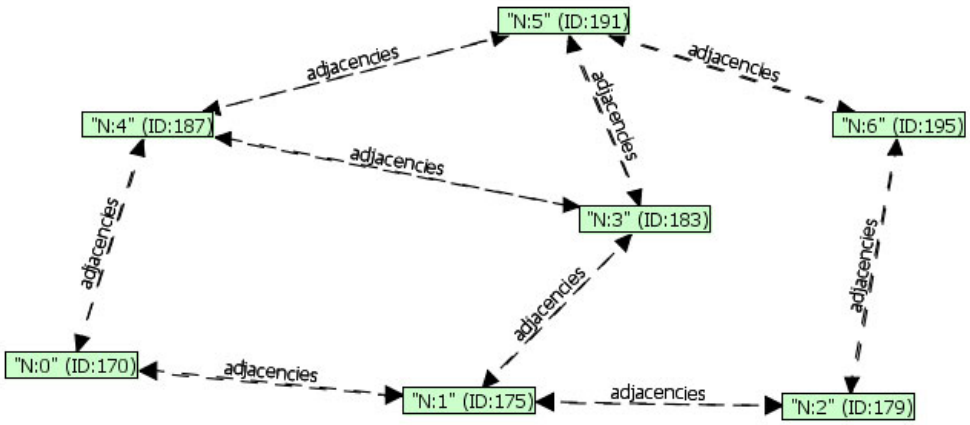
Completamento rotazione destra



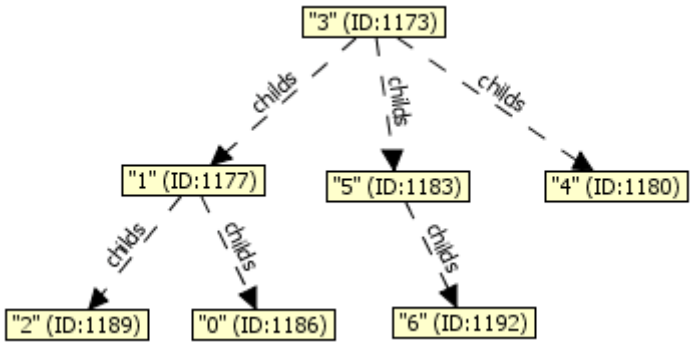
Visualizzazione albero AVL completo con 14 nodi

Algoritmo di Dijkstra

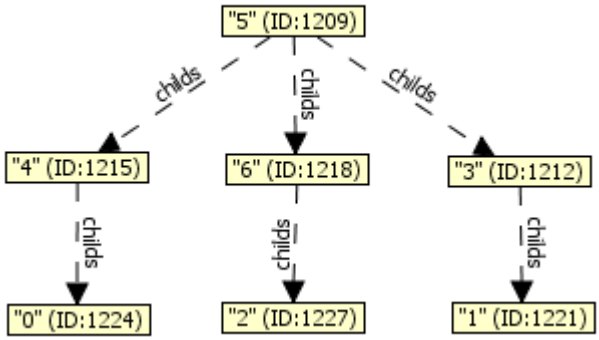
E' stato realizzato un algoritmo di esempio che risolve il problema del *Single Source Shortest Path* (SSSP) su un grafo semplice e connesso, con valori agli spigoli unitari e rappresentato con una matrice di adiacenza. E' stato utilizzato il classico algoritmo di Dijkstra[A6] e si riporta il codice di esempio in appendice. L'algoritmo restituisce l'albero dei cammini minimi a partire da un nodo dato; è stata inoltre sviluppata una struttura dati che trasforma una matrice di adiacenza in un grafo a oggetti, in modo da permetterne la visualizzazione intuitiva in AlgoExplorer. Vengono riportate di seguito alcune schermate dell'area di visualizzazione di AlgoExplorer dati diversi input di nodi di partenza.



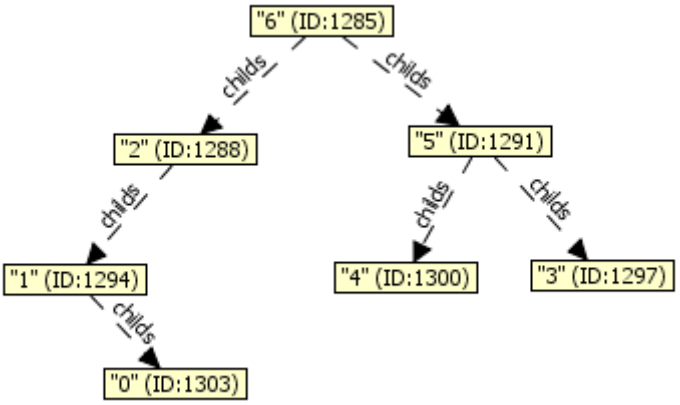
Visualizzazione del grafo usato nel caso di esempio



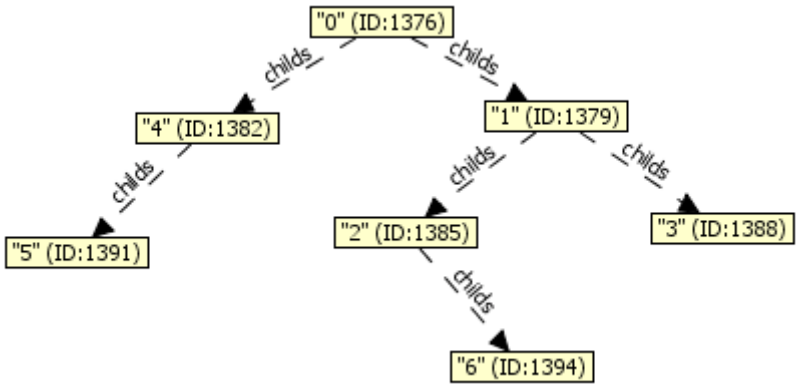
Albero dei cammini minimi generato a partire dal nodo "3"



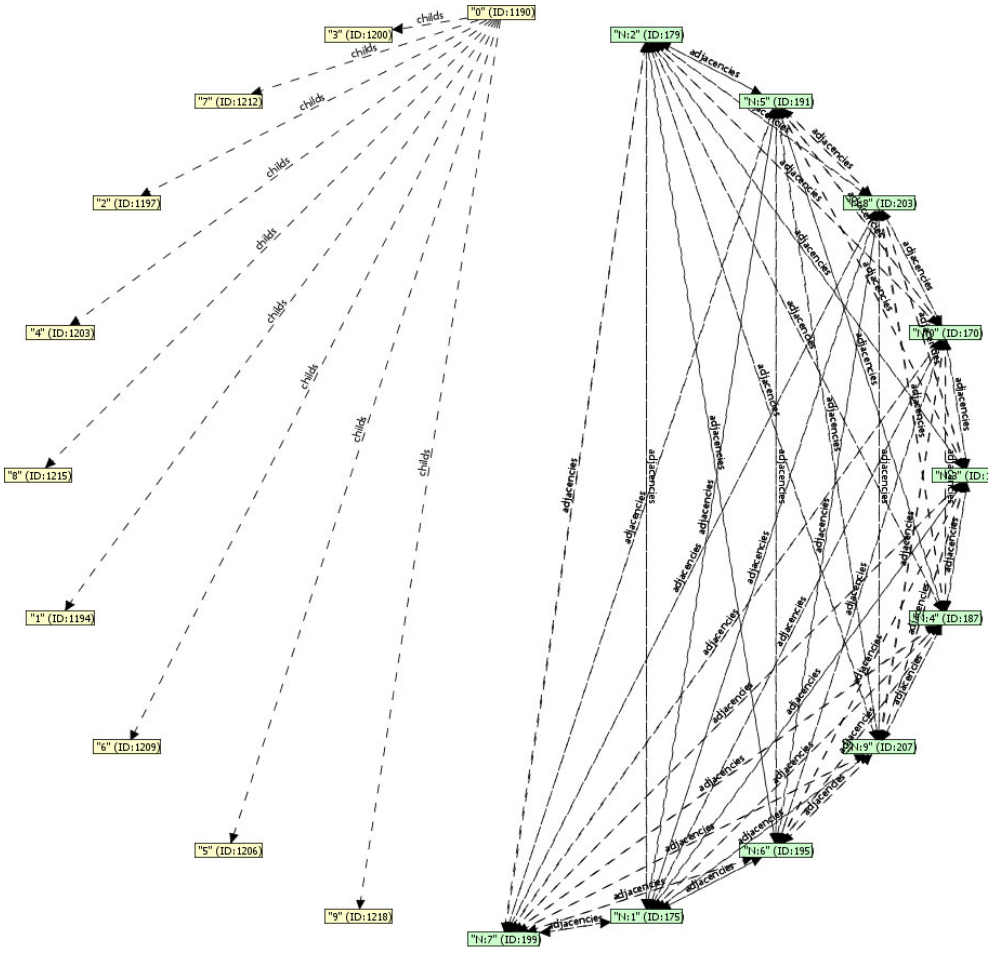
Albero dei cammini minimi generato a partire dal nodo "5"



Albero dei cammini minimi generato a partire dal nodo "6"



Albero dei cammini minimi generato a partire dal nodo "0"



Esempio di visualizzazione di un grafo, con 10 nodi, completamente connesso e di un albero dei cammini minimi estratto a partire da un nodo

Confronti con applicazioni simili

AlgoExplorer, ovviamente, non è la sola applicazione che si occupa della visualizzazione grafica delle strutture dati manipolate da algoritmi. Si ritiene quindi opportuno, a questo punto, analizzare e confrontare con AlgoExplorer, almeno altre due applicazioni che affrontano lo stesso tipo di visualizzazione.

Le applicazioni sono:

- The Lightweight Java Visualizer (LJV) [A7] [B16]
- jGRASP [A8] [B17]

LJV è un semplice tool per la visualizzazione di strutture dati in Java. L'applicazione usa la Java Reflection per l'introspezione dei dati e la libreria Graphviz[B18] per la visualizzazione grafica del grafo che rappresenta le strutture dati esaminate. LJV non possiede nessuna interfaccia grafica e lo sviluppatore dell'algoritmo, che si desidera analizzare, deve integrare le chiamate alla libreria LJV per permetterne l'interazione. Output di LJV è un file di descrizione testuale del grafo, che poi sarà disegnato dalla libreria Graphviz.

JGRASP è un leggero ambiente di sviluppo, implementato interamente in Java e creato anche per la visualizzazione automatica delle strutture dati a fini didattici. JGRASP permette lo sviluppo dei programmi direttamente dalla sua GUI e in fase di esecuzione di questi ultimi permette, tramite un'interfaccia abbastanza complessa, di visualizzare le strutture dati sviluppate e di riconoscere e visualizzare correttamente e in modo intuitivo le tradizionali strutture dati come stack, code, liste collegate, alberi e hashtable. Anche jGRASP come AlgoExplorer usa JDI per l'introspezione delle strutture dati.

AlgoExplorer, invece, è un'applicazione specificatamente concepita per la visualizzazione delle strutture dati che entrano in gioco nell'esecuzione dei programmi, integra, a differenza di LJV e come jGRASP, un proprio ambiente di visualizzazione grafica e non costringe il programmatore, come fa anche jGRASP ma non LJV, a rispettare specifiche regole o utilizzare particolari librerie. Tuttavia jGRASP è anche un ambiente di sviluppo abbastanza complesso al primo approccio, AlgoExplorer invece è semplice, usabile e limitandosi alla visualizzazione delle strutture dati, velocizza e rende pratico il suo utilizzo, anche se non permette, nella sua attuale versione, il riconoscimento e la visualizzazione in modo intuitivo di strutture dati tradizionali come ad esempio gli array.

Conclusioni e sviluppi futuri

Il lavoro svolto nello studio preliminare di fattibilità, nella progettazione e nello sviluppo di AlgoExplorer, ha messo in evidenza i punti di forza dell'approccio utilizzato.

AlgoExplorer risulta essere semplice, usabile e realizza completamente gli obiettivi posti dai requisiti utente richiesti e descritti all'inizio di questa relazione.

Da quanto analizzato nel capitolo che ha studiato gli esempi d'uso su algoritmi conosciuti, si può assegnare ad AlgoExplorer una validità didattica, perché facilmente permette lo studio e la comprensione anche di algoritmi complessi.

L'utilizzazione di JDI ha reso possibile un'introspezione priva di interferenze dei programmi eseguiti da AlgoExplorer; inoltre questa procedura potrebbe, nelle versioni successive di AlgoExplorer, essere utilizzata anche per visualizzare in modo intuitivo e automatico, magari con un layout personalizzato, strutture dati che per ora non sono visualizzate in questo modo (es: array). JDI tuttavia, a causa dei numerosi eventi che interrompono l'esecuzione dei programmi gestiti da AlgoExplorer, eventi che sono necessari ai

fini dell'introspezione delle strutture dati, così come anche descritto in [A8] per jGRASP, rallenta considerevolmente l'esecuzione; pertanto AlgoExplorer risulta meglio adatta a studiare semplici programmi come sono ad esempio molti algoritmi, piuttosto che complesse applicazioni Java.

AlgoExplorer potrebbe essere integrato successivamente in un IDE (Integrated Development Environment) quale ad esempio Eclipse[B7], per permettere la visualizzazione e l'esecuzione in modo diretto degli algoritmi nel momento stesso in cui lo sviluppatore progetta il codice. Questa soluzione potrebbe essere sviluppata semplicemente permettendo il lancio dell'applicazione in esame dall'IDE utilizzata o integrandola in maniera più incisiva con quest'ultima, precaricando ad esempio il codice sorgente e le classi binarie sviluppate nell'ambiente di lavoro dell'IDE utilizzato.

Risulterebbe poi utile sviluppare una funzione di salvataggio delle impostazioni e delle scelte di visualizzazione effettuate su un workspace; in questo modo queste impostazioni potrebbero essere facilmente preconfigurate, distribuite e successivamente caricate permettendo una facile visualizzazione didattica e automatica degli algoritmi di interesse.

AlgoExplorer nelle sue versioni successive, potrebbe permettere di generare, tramite il prodotto open source Adobe FLEX SDK[B19], delle animazioni del suo output grafico, questo per visualizzare il funzionamento degli algoritmi per scopo didattico anche su pagine

web; in tal caso si potrebbe dare la possibilità all'utente creatore dell'animazione, di "fotografare" istanti di interesse di visualizzazione e aggiungere un commento testuale ad ogni schermata, che possa spiegare gli aspetti teorici dell'algoritmo analizzato; in questo modo, per realizzare l'animazione, AlgoExplorer potrebbe esportare un file *MXML*[B20] testuale che poi può essere compilato con FLEX SDK per la generazione dell'animazione, visualizzabile su qualsiasi browser provvisto di plug-in Flash Player[B21] (supportato dalla maggior parte dei sistemi operativi: Windows, Mac e Linux).

Appendice

Codice sorgente degli algoritmi per gli esempi d'uso

```
 1 package test;
 2
 3 import java.util.List;
 4 import java.util.Vector;
 5
 6 /**
 7  *
 8  * Per implementare il metodo sssp, ho usato
 9  * il classico algoritmo di Dijkstra,
10  * direttamente usando la matrice di adiacenza data,
11  * e non effettuando nessun side-effect su questa;
12  * per calcolare il minimo sui nodi ho usato una coda
13  * di priorita' indiretta che usa multiway heap e che
14  * restituisce direttamente l'indice dell'elemento minimo
15  * dell'array, permettendo di aggiornare le priorita'
16  * dello stesso array dinamicamente.
17  * La coda di priorita' usata e' desunta dal
18  * programma 20.10 di "Algoritmi in Java" di R.Sedgewick.
19  *
20  * @author      Pier Paolo Ciarravano
21  */
22 public class Dijkstra {
23
24     public static void main(String[] args) {
25
26         double[][] graph = {
27             {0.0d, 1.0d, 0.0d, 0.0d, 1.0d, 0.0d, 0.0d},
28             {1.0d, 0.0d, 1.0d, 1.0d, 0.0d, 0.0d, 0.0d},
29             {0.0d, 1.0d, 0.0d, 0.0d, 0.0d, 0.0d, 1.0d},
30             {0.0d, 1.0d, 0.0d, 0.0d, 1.0d, 1.0d, 0.0d},
31             {1.0d, 0.0d, 0.0d, 1.0d, 0.0d, 1.0d, 0.0d},
32             {0.0d, 0.0d, 0.0d, 1.0d, 1.0d, 0.0d, 1.0d},
33             {0.0d, 0.0d, 1.0d, 0.0d, 0.0d, 1.0d, 0.0d}
34         };
35
36         GraphNode<String, Double> graphNode =
37                 GraphNode.getGraphFromMatrix(graph);
38
39         TreeNode<Integer> test1 = sssp(graph, 3);
40         TreeNode<Integer> test2 = sssp(graph, 2);
41         TreeNode<Integer> test3 = sssp(graph, 5);
42         TreeNode<Integer> test4 = sssp(graph, 1);
43
44
45     }
46
47     public Dijkstra() {
```

```
48      }
49
50      /*
51       * Il metodo sssp riceve in input una
52       * matrice double[][] eam che rappresenta
53       * la matrice di adiacenza estesa di un
54       * grafo semplice, connesso e pesato
55       * sugli spigoli, un intero int i,
56       * con 0<=i<n = eam.length,
57       * che rappresenta l'indice della radice
58       * dell'albero dei cammini minimi da costruire
59       * e restituisce la radice un albero
60       * dei cammini minimi a partire dal vertice i.
61       * La matrice in input è simmetrica
62       * rispetto alla diagonale principale,
63       * ove sono presenti solo zeri, non contiene
64       * numeri negativi e descrive
65       * un grafo sicuramente connesso.
66       */
67      public static TreeNode<Integer> sssp(double[][] eam,
68              int i) {
69          //Salto i controlli su eam
70
71          //mi limito ad un controllo su null
72          if (eam == null) {
73              return null;
74          }
75          //Controllo la validita dell'indice i
76          if ((i < 0) || (i >= eam.length)) {
77              return null;
78          }
79          //i = i-1;
80          //int[] pred = new int[eam.length];
81          @SuppressWarnings("unchecked")
82          TreeNode<Integer>[] nodes =
83                  new TreeNode[eam.length];
84
85          //boolean[] mark = new boolean[eam.length];
86          double[] wt = new double[eam.length];
87          int j;
88          int u = 0;
89          double p = 0.0d;
90
91          //Inizializzazione
92          IndirectPQ pQ = new IndirectPQ(eam.length, wt);
93          for (j = 0; j < eam.length; j++) {
94              //mark[i]=false;
95              wt[j] = Double.MAX_VALUE;
96              pQ.insert(j);
97          //pred[j] = 0;
98          }
99          wt[i] = 0.0d; //Nodo di partenza, distanza 0
100
101         pQ.lower(i);
102         //pred[i] = -1; //-1 per radice
```

```
103         TreeNode<Integer> root =
104                 new TreeNode<Integer>(i);
105         nodes[i] = root;
106 
107         while (!pQ.empty()) {
108             u = pQ.getmin();
109             // mark[u] = true;
110             for (j = 0; j < eam.length; j++) {
111                 if (eam[u][j] > 0.0d) {
112                     // if (mark[i]) {
113                     p = wt[u] + eam[u][j];
114                     if (wt[j] > p) {
115                         wt[j] = p;
116                         pQ.lower(j);
117                         // pred[j] = u+1;
118                         if (nodes[j] == null) {
119                             TreeNode<Integer> node =
120                                 new TreeNode<Integer>(j);
121 
122                             //Utilizzando parent
123                             //node.parent = nodes[u];
124 
125                             //Utilizzando childs
126                             nodes[u].addChild(node);
127 
128                             nodes[j] = node;
129                         } else {
130                             //Utilizzando parent
131                             //nodes[j].parent = nodes[u];
132 
133                             //Utilizzando childs
134                             nodes[u].addChild(nodes[j]);
135                         }
136                     }
137                 // }
138                 }
139             }
140         }
141 
142         //return pred;
143         return root;
144 
145     }
146 }
147 
148 /*
149  * Classe TreeNode
150  *
151  * @author      Pier Paolo Ciarravano
152  */
153 class TreeNode<T> {
154 
155     T element;
156     TreeNode<T> firstChild;
157     TreeNode<T> nextSibling;
```

```
158     TreeNode<T> parent;
159     List<TreeNode<T>> childs;
160 
161     public TreeNode(T theElement) {
162         element = theElement;
163 
164         childs = new Vector<TreeNode<T>>(0);
165     }
166 
167     public void addChild(TreeNode<T> child) {
168         this.childs.add(child);
169     }
170 }
171 
172 /*
173  * Algoritmo da:
174  *
175  * "Sedgewick, R. and Schidlowsky, M. 1998
176  * Algorithms in Java, Third Edition, Parts 1-4:
177  * Fundamentals, Data Structures, Sorting,
178  * Searching. 3rd.
179  * Addison-Wesley Longman Publishing Co., Inc."
180  *
181  * http://www.cs.princeton.edu/~rs/Algs3.java1-4/code.txt
182  */
183 class IndirectPQ {
184 
185     private int N;
186     private final int d = 3;
187     private final int dd = d - 2;
188     private double[] a;
189     private int[] pq,  qp;
190 
191     private boolean less(int i, int j) {
192         return a[pq[i]] < a[pq[j]];
193     }
194 
195     private void exch(int i, int j) {
196         int t = pq[i];
197         pq[i] = pq[j];
198         pq[j] = t;
199         qp[pq[i]] = i;
200         qp[pq[j]] = j;
201     }
202 
203     private void swim(int k) {
204         while (k > 1 && less(k, (k + dd) / d)) {
205             exch(k, (k + dd) / d);
206             k = (k + dd) / d;
207         }
208     }
209 
210     private void sink(int k, int N) {
211         int j;
212         while ((j = d * (k - 1) + 2) <= N) {
```

```
213             for (int i = j + 1; i < j + d && i <= N; i++)
214             {
215                 if (less(i, j)) {
216                     j = i;
217                 }
218             }
219             if (!(less(j, k))) {
220                 break;
221             }
222             exch(k, j);
223             k = j;
224         }
225     }
226 
227     IndirectPQ(int maxN, double[] a) {
228         this.a = a;
229         this.N = 0;
230         pq = new int[maxN + 1];
231         qp = new int[maxN + 1];
232         for (int i = 0; i <= maxN; i++) {
233             pq[i] = 0;
234             qp[i] = 0;
235         }
236     }
237 
238     boolean empty() {
239         return N == 0;
240     }
241 
242     void insert(int v) {
243         pq[++N] = v;
244         qp[v] = N;
245         swim(N);
246     }
247 
248     int getmin() {
249         exch(1, N);
250         sink(1, N - 1);
251         return pq[N--];
252     }
253 
254     void lower(int k) {
255         swim(qp[k]);
256     }
257 }
258 
259 /*
260  * Classe GraphNode
261  *
262  * @author      Pier Paolo Ciarravano
263  */
264 class GraphNode<T, W> {
265 
266     private T element;
267     private List<GraphNode<T, W>> adjacencies;
```

```
268     private List<W> weights;
269
270     public GraphNode(T element) {
271         this.element = element;
272         adjacencies = new Vector<GraphNode<T, W>>(0);
273         weights = new Vector<W>(0);
274     }
275
276     public T getElement() {
277         return element;
278     }
279
280     public void setElement(T element) {
281         this.element = element;
282     }
283
284     public List<GraphNode<T, W>> getAdjacencies() {
285         return adjacencies;
286     }
287
288     public List<W> getWeights() {
289         return weights;
290     }
291
292     public void addAdjacency(GraphNode<T, W> graphNode) {
293         addAdjacency(graphNode, null);
294     }
295
296     public void addAdjacency(GraphNode<T, W> graphNode,
297             W weight) {
298         this.adjacencies.add(graphNode);
299         this.weights.add(weight);
300     }
301
302     //Crea un nodo lo connette e lo restituisce
303     public GraphNode<T, W> addAdjacency(T element,
304             W weight) {
305         GraphNode<T, W> graphNode =
306                 new GraphNode<T, W>(element);
307         this.adjacencies.add(graphNode);
308         this.weights.add(weight);
309
310         return graphNode;
311     }
312
313     public static GraphNode<String, Double>
314             getGraphFromMatrix(double[][] matrix) {
315         GraphNode<String, Double>[] nodi =
316                 new GraphNode[matrix.length];
317         for (int i = 0; i < matrix.length; i++) {
318             nodi[i] = new GraphNode<String, Double>
319                     ("N:" + (i));
320         }
321
322         for (int i = 0; i < matrix.length; i++) {
```

```
323             for (int j = 0; j < matrix.length; j++) {
324                 if (matrix[i][j] != 0) {
325                     nodi[i].addAdjacency(nodi[j],
326                             matrix[i][j]);
327                 }
328             }
329         }
330 
331         return nodi[0];
332     }
333 }
334 
```

Bibliografia

[A1] Baecker, R.M. (1998). Sorting Out Sorting: A Case Study of Software Visualization for Teaching Computer Science. In Stasko, J., Domingue, J., Brown, M., and Price, B. (Eds.), Software Visualization: Programming as a Multimedia Experience. MIT Press, 369-381

[A2] M. H. Brown and R. Sedgewick. A system for algorithm animation. In SIGGRAPH '84: Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, pages 177–186, New York, NY, USA, 1984. ACM Press.

[A3] Binder, W., Hulaas, J., and Moret, P. 2007. Reengineering Standard Java Runtime Systems through Dynamic Bytecode Instrumentation. In Proceedings of the Seventh IEEE international Working Conference on Source Code Analysis and Manipulation (September 30 - October 01, 2007). SCAM. IEEE Computer Society, Washington, DC, 91-100.

[A4] Bailey, D. A. 2001 Java Structures: Data Structures in Java for the Principled Programmer. 2nd. McGraw-Hill, Inc.

[A5] Weiss, M. A. 2005 Data Analysis and Algorithm Analysis in Java (2nd Edition). Addison-Wesley Longman Publishing Co., Inc.

[A6] Creak, A. 2002. Edsger W. Dijkstra. SIGPLAN Not. 37, 12 (Dec. 2002), 14-16. DOI=http://doi.acm.org/10.1145/636517.636521

[A7] Hamer, J. 2004. Visualising Java data structures as graphs. In Proceedings of the Sixth Conference on Australasian Computing Education - Volume 30 (Dunedin, New Zealand). R. Lister and A. Young, Eds. ACM International Conference Proceeding Series, vol. 57. Australian Computer Society, Darlinghurst, Australia, 125-129.

[A8] Cross, J., Hendrix, T., Umphress, D., Barowski, L., Jain, J., and Montgomery, L. 2009. Robust Generation of Dynamic Data Structure Visualizations with Multiple Interaction Approaches. Trans. Comput. Educ. 9, 2 (Jun. 2009), 1-32. DOI= http://doi.acm.org/10.1145/1538234.1538240

Sitografia

[B1] Baecker, R.M., with assitance of Dave Sherman. 30 minute colour sound film, Dynamic Graphics Project, University of Toronto, 1981. (Excerpted and "reprinted" in SIGGRAPH Video Review 7, 1983.) (Distributed by Morgan Kaufmann, Publishers.):

`http://www.kmdi.utoronto.ca/rmb/video/sos_recap.mov`

`http://www.kmdi.utoronto.ca/rmb/video/sos_dotclouds.mov`

[last visit: 6 Jen. 2010]

[B2] Data Structure and Algorithm Visualization Library for Computer Science Education:

`http://wiki.algoviz.org`

[last visit: 6 Jen. 2010]

[B3] Java SE Downloads - Sun Developer Network (SDN)

`http://java.sun.com/javase/downloads/index.jsp`

[last visit: 6 Jen. 2010]

[B4] Java Debug Interface (JDI)

`http://java.sun.com/javase/6/docs/jdk/api/jpda/jdi/index.html`

[last visit: 6 Jen. 2010]

[B5] Apache Byte Code Engineering Library

`http://jakarta.apache.org/bcel/`

[last visit: 6 Jen. 2010]

[B6] JGraph

`http://www.jgraph.com/jgraph5.html`

[last visit: 6 Jen. 2010]

[B7] Eclipse IDE

`http://www.eclipse.org/`

[last visit: 6 Jen. 2010]

[B8] Java Reflection API

`http://java.sun.com/docs/books/tutorial/reflect/index.html`

[last visit: 6 Jen. 2010]

[B9] Java Virtual Machine Tool Interface (JVMTI)

`http://java.sun.com/javase/6/docs/technotes/guides/jvmti/index.html`

[last visit: 6 Jen. 2010]

[B10] Bytecode Instrumentation (BCI)

`http://java.sun.com/j2se/1.5.0/docs/api/java/lang/instrument/package-summary.html`

`http://weblogs.java.net/blog/kellyohair/archive/2005/05/bytecode_instru_1.html`

[last visit: 6 Jen. 2010]

[B11] Java Platform Debugger Architecture (JPDA)

`http://java.sun.com/javase/6/docs/technotes/guides/jpda/index.html`

[last visit: 6 Jen. 2010]

[B12] Apache Log4J

`http://logging.apache.org/log4j/1.2/index.html`

[last visit: 6 Jen. 2010]

[B13] Java VisualVM

`https://visualvm.dev.java.net/`

[last visit: 6 Jen. 2010]

[B14] Java Data Structures Package by Duane A. Bailey

`http://www.cs.williams.edu/~bailey/JavaStructures/Software.html`

[last visit: 6 Jen. 2010]

[B15] Code of Data Analysis and Algorithm Analysis in Java - Mark Allen Weiss

`http://users.cis.fiu.edu/~weiss/dsaajava2/code/`

[last visit: 6 Jen. 2010]

[B16] The Lightweight Java Visualizer (LJV)

`http://www.cs.auckland.ac.nz/~j-hamer/LJV.html`

[last visit: 6 Jen. 2010]

[B17] jGRASP - An Integrated Development Environment with Visualizations for Improving Software Comprehensibility

`http://www.jgrasp.org`

[last visit: 6 Jen. 2010]

[B18] Graphviz - Graph Visualization Software

`http://www.graphviz.org`

[last visit: 6 Jen. 2010]

[B19] Adobe FLEX SDK

`http://opensource.adobe.com/wiki/display/flexsdk/Flex+SDK`

[last visit: 6 Jen. 2010]

[B20] Adobe FLEX MXML Language

`http://learn.adobe.com/wiki/display/Flex/MXML`

`http://www.adobe.com/devnet/flex/quickstart`
`/coding_with_mxml_and_actionscript/`

[last visit: 6 Jen. 2010]

[B21] Adobe Flash Player

`http://www.adobe.com/products/flashplayer/`

[last visit: 6 Jen. 2010]