Design and Implementation of the JGraph Swing Component

Gaudenz Alder alder@jgraph.com

1.0.6,28.February 2002

Versione Italiana di Francesco Candeliere francesco.candeliere@tiscali.it

14 February 2003

Abstract

Today's user interface (UI) libraries offer components for lists,tables,and trees,but graph components are rarely implemented.JGraph provides a fully standards-compliant graph component for the Java Swing UI library that supports extended display and editing options.

This paper provides provides a description of the JGraph component.The document is structured into modules,and illustrated using UML and other diagrams.The study will outline JGraph's design and implementation with focus on Swing compatibility,and explain where it was necessary to extend Swing design.

The target readers are practitioners who need a definition of JGraph's architecture.The document provides an in-depth discussion of the component's design and implementation,and thoroughly explains the ideas and concepts behind the architecture.

Indice

1 Introduzione

Dopo una breve introduzione ai fondamenti della teoria dei grafi e di Swing, questa introduzione sottolinea la struttura di questo documento, e la letteratura che è stata usata. Per informazioni addizionali, aggiornamenti, codice binario e sorgente, si veda la home page di Jgraph a http://www.jgraph.com.

1.1 Panoramica

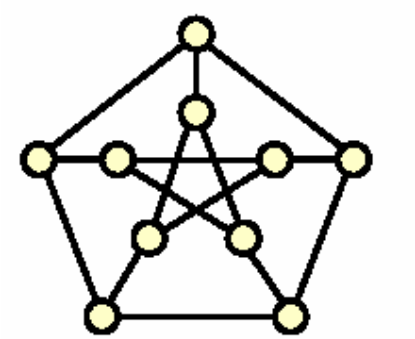

Fig. 1: The Peterson graph

L'intenzione di questo progetto è di fornire una implementazione disponibile liberamente, che supporti Swing, di un componente per grafi. Essendo un componente Swing per grafi, Jgraph è basato sulla teoria matematica dei grafi e sulla libreria per interfacce utenti (UI) Swing. Combinando le due, un componente Swing di UI per visualizzare grafi è ottenuto. (Il termine grafo è usato nella eccezione della teoria dei grafi in questo documento, non va confuso con la funzionalità di plot.)

I seguenti principi hanno guidato l'implementazione di Jgraph :

- Piena compatibilità con Swing
- Design chiaro ed efficiente
- Tempo di download piccolo
- 100% puro Java

L'idea dietro questi principi è basata sulla esperienza in altre librerie, che tipicamente diventano così grosse e complesse, da essere non più standard.
In Jgraph invece, l'architettura di base è la stessa dei componenti standard di Swing e i metodi e i nomi delle variabili seguono lo standard Java/Swing. Ciò ha il vantaggio di ridurre i tempi di apprendimento e di permettere il riuso di codice esistente.

1.1.1 Teoria dei Grafi

Il concetto di grafo è basato sulla ben nota teoria matematica, rigorosamente definita in [1],[2],[3]. Un grafo consiste di vertici e di spigoli che connettono coppie di vertici. L'esatto pattern geometrico non è specificato dalla teoria, come ovvio che sia.

Essendo una generalizzazione matematica delle lista e degli alberi, i grafi sono utili quando le liste e gli alberi non sono più sufficenti per modellare relazioni. Per esempio, in un albero ciascun nodo ha al massimo un padre, e zero o più figli, mentre in un grafo, ciascun vertice semplicemente ha zero o più vicini. (Nel caso di una lista, ciascun nodo ha al massimo due vicini).

Il termine *vicini* serve per enfatizzare un contesto più ampio di quello degli alberi, ove vi sono solo padri e figli. Il termine *cella* è usato al posto di *nodo* in questo documento per distinguere tra gli elementi base di un grafo da quelli di liste ed alberi.
Formalmente, un grafo consiste di un insieme (set) non vuoto di elementi *V(G)* e di un sotto set *E(G)* di coppie non ordinate di elementi distinti di *V(G)*. Gli elementi di *V(G)*, chiamati vertici di G, possono essere rappresentati da punti. Se (x,y) appartengono a *E(G)*, allora lo spigolo (x,y) può essere rappresentato da un arco che congiunge x ad y.
Allora x ed y si dicono *adiacenti* e lo spigolo (x,y) è adiacente con x e y. Se (x,y) non è uno spigolo, allora i vertici x e y si dicono *non adiacenti*.

La teoria dei grafi inoltre offre algoritmi standard per risolvere problemi comuni. Per esempio, l'algoritmo di *Dijkstra* può essere usato per trovare il percorso più breve tra due vertici del grafo. L'algoritmo di *Dijkstra* e moltri altri algoritmi standard per i grafi posso trovarsi in [4],[5],[6]. (Una implementazione dell'algoritmo di *Dijkstra* si trova in un esempio di Jgraph).

Per riassumere, i grafi sono usati come un paradigma in Jgraph per visualizzare ogni tipo di rete di oggetti collegati. Una strada in una rete di computer, una molecola, architetture del software o schemi di database sono esempi di grafo. Poiché i grafi sono generalizzazioni matematiche di alberi e liste, Jgraph può anche essere usato per visualizzare, per esempio, un albero del file system.

1.1.2 Swing

Swing è la libreria per interfacce utenti di Java e fornisce gli elementi che possono essere piazzati sulla finestra di una aplicazione. Tali elementi sono chiamati componenti dell'interfaccia o semplicemente *componenti*. Componenti comuni sono pulsanti, liste, alberi e tabelle.

Swing si basa su AWT (Abstract Windowing Toolkit) che è un'altra libreria per interfacce utenti (UI) che si trova nella JFC di Java e può essere usata indipendentemente da Swing. Swing, dal canto suo, è più sofisticata e fornisce più funzionalità.

Una buona introduzione a Java e AWT si trova in [7],[8]. I libri [9] e [10] sono per programmatori esperti. A differenza di AWT, Swing è spiegato in [11], [12] e [13].
I documenti di cui sopra, discutono i componenti esistenti in profondo dettaglio, ma non spiegano come crearne di nuovi.

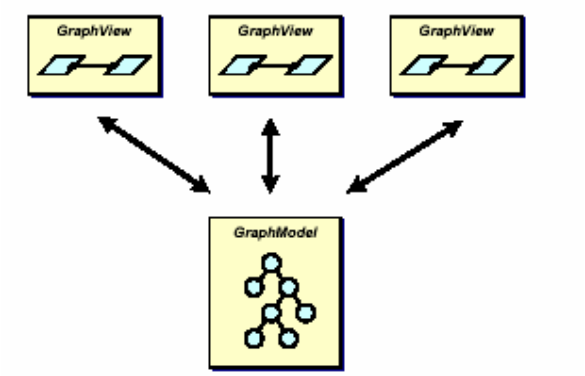

Fig. 2: Model-View-Control (MVC)

In poche parole spieghiamo il modello MVC di Swing, che è necessario conoscere, non solo per scrivere nuovi componenti Swing, ma anche per utilizzare al meglio Jgraph.
Il *modello* fornisce i dati, che sono un grafo, nel senso usato nella teoria dei grafi, e l'oggetto Jgraph fornisce una *vista* dei dati, in un modo dipendente dalla piattaforma. Il design pattern MVC è usato in Swing per separare il modello dalla vista e per stabilire una relazione tra loro, di modo che, più di una vista possa essere attaccata allo stesso modello, e che tutte le viste siano automaticamente aggiornate quando vi sono cambiamenti nel modello.

Jgraph è largamente basato sul componente Swing per gli alberi, chiamato Jtree, spiegato in [14]. Alcune idee derivano dai componenti text di Swing, di preciso dall'intefaccia **Element**, che è discussa in [15] e dall'intefaccia **View** che è descritta in [16].
La stessa classe **Jgraph** è una estensione di **Jcomponent** che è la classe base di tutti i componenti Swing.

Jgraph adotta tutti gli standard Swing, come quelli per il look and feel (L&F) rimpiazzabile, per il trasferimento dei dati, per l'accessibilità, per l'internazionalizzazione e per la serializzazione. In più tutti i concetti usati in Jgraph sono in comune ai ben noti concetti di Swing. Per caratteristiche più evolute, quali supporti per undo/redo e XML, sono ancora adottati gli standard Swing. Affianco a ciò Jgraph adotta le convenzioni Java sui nomi di variabile, sul layout del codice sorgente e sulla documentazione con javadoc.

8 References

[1] Biggs.Discrete Mathematics (Revised Edition).Oxford University Press,New York NY,1989.
[2] Aigner.Diskrete Mathematik.Vieweg,Braunschweig/Wiesbaden, 1993.
[3] Ottmann,Widmayer.Algorithmen und Datenstrukturen (2.Auflage). BI-Wiss.-Verl.,Mannheim,Leipzig,Wien,Zürich.1993
[4] Sedgewick.Algorithmen.Addison-Wesley,Reading MA,1991.
[5] Nievergelt,Hinrichs.Algorithms and data structures.Prentice-Hall, Englewood Cliffs NJ,1993.
[6] Harel.Algorithmics.Addison-Wesley,Reading MA,1992.
[7] Bishop.Java Gently.Addison-Wesley,Reading MA,1998.
[8] Jackson,McClellan.Java 1.2 by example.Sun Microsystems,Palo Alto CA,1999.
[9] Flanagan.Java in a nutshell (2nd Edition).O'Really & Associates, Sebastopol CA,1997.
[10] Stärk,Schmid,Börger.Java and the Java Virtual Machine. Springer,Heidelberg,2001.
[11] Geary.Graphic Java 2,Volume II: Swing (3rd Edition).Sun Microsystems,Palo Alto CA,1999.
[12] Geary.Graphic Java 2,Volume III: Advanced Swing (3rd Edition). Sun Microsystems,Palo Alto CA,1999 (?).
[13] The Swing Tutorial.
http://java.sun.com/docs/books/tutorial/index.html
[14] JTree API Specification.
http://java.sun.com/j2se/1.4/docs/api/javax/swing/JTree.html
[15] Violett.The Element Interface.
http://java.sun.com/products/jfc/tsc/articles/text/element_interface/
[16] View Interface API Specification.
http://java.sun.com/j2se/1.4/docs/api/javax/swing/text/View.html
[17] Gamma,Helm,Johnson,Vlissides.Design Patterns.Addison-Wesley,Reading MA,1995.
[18] Alder.The JGraph Tutorial.
http://jgraph.sourceforge.net/tutorial.html
[19] The Java 1.4 API Specification.
http://java.sun.com/j2se/1.4/docs/api/
[20] Alder.The JGraph 1.0 API Specification.http://api.jgraph.com
[21] Alder.JGraph.Semester Work,Department of Computer Science, ETH Zürich,Switzerland,2001.

1.2 Decomposizione di Jgraph

La decomposizione di Jgraph in moduli, divide la definizione globale in una serie di sottodefinizioni, trattabili singolarmente. Seguendo il design pattern MVC, il framework è diviso in un modello, una vista e un controllore. Questi capitoli sono preceduti da una panoramica del componente stesso.

La panoramica del componente compara Jgraph a Jtree e sottolinea le differenze principali e la conseguenze riscontrabili nelle API. In più discute le API con un occhio di riguardo al pattern MVC, alla sua implementazione in Jgraph e all'utilizzo di viste multiple.
La parte sul modello descrive l'interfaccia di modello sottostante, il modello di selezione e gli elemneti ivi contenuti. Di più, vengono illustrate le classi usate per manipolare il modello.

La parte sulla vista studia la rappresentazione interna di ciò che del grafo sarà visualizzato, e come avviene il mappaggio tra il modello e le viste. Un'attenzione particolare è data al così detto pattern geometrico o layout del grafo e a quello che è definito contesto (context) del grafo.

La parte sul controllo spiega il processo di rendering, i passi necessari per l'editing in-place e per la gestione delle celle, e gli oggetti coinvolti nel trasferimento dei dati e per la selezione marquee.

Il modello ad eventi è spiegato in un capito addizionale, che sottolinea i meccanismi di aggiornamento e di undo, oltre agli aspetti dinamici del sistema.

Il capitolo finale riassume i principali risultati, trae delle conclusioni circa l'implementazione corrente e discute possibili future espansioni di Jgraph.

1.3 Sorgenti e Letteratura

Il componente Jgraph è ampiamente basato su [14]. Alcune idee vengono da [15]. La descrizione dei design pattern usati è in [17].

Il libro standard su Swing è di certo [11] ed è una buona base per capire questo documento. Alcune implementazioni di algoritmi specifici per grafi sono basate su [4] e [5]. I lettore dovrebbe essere famigliare con i concetti più importanti della teoria dei grafi, che si possono trovare in [1].

In aggiunta a questo documento è disponibile un tutorial [18] e una definizione completa della API [20], che si trovano sul sito http://www.jgraph.com

2 Il Componente Jgraph

L'implementazione di Jgraph è interamente basata sul codice sorgente della classe Jtree [14], che è il componente base per la visualizzazione di alberi in Swing. Invece di spiegare Jgraph in poche parole, questa descrizione si focalizza sulle differenze esistenti tra Jgraph e Jtree.
Alcune caratteristiche standard di Swing come serializzazione, datatransfer e pattern MVC sono spiegate in appendice.

2.1 Fondamenti

Jgraph non è una estensione di Jtree in senso stretto; piuttosto, è una modifica del codice sorgente di Jtree. Nel seguito, i cambiamenti sono spiegati sommariamente, sottolineando le classi e i metodi modificati e/o introdotti ex-novo. Le modifiche sono raggruppate in :

- Differenze tra alberi e grafi
- Caratteristiche di Jgraph

Le seguenti importanti differenze tra alberi e grafi fanno da fondamento al componente Jgraph :

2.2.1 Celle Sovrapponibili

In un albero, la posizione e la visibilità di una cella dipendono dal suo stato di espansione.In un grafo invece, la posizione e la grandezza di una cella è definita dall'utente, con la possibilità che le celle si sovrappongano. Di conseguenza, è fornita una modalità per enumerare le celle che intersecano la stessa posizione e per modificare l'ordine in cui queste sono ritornate.

2.1.2 Celle di Più Tipi

Un albero consiste di nodi, mentre un grafo consiste di tipi di celle multipli. Di conseguenza, Jgraph fornisce una vista specifiche per ogni tipo di cella, attraverso un renderer, un editor e un gestore (handler). In più, fornisce una vista globale del grafo (graph view), che mantiene la propria rappresentazione interna del grafo. L'idea della viste è stata adottata da componente per il testo di Swing [16].

2.1.3 Geometrie Separate

La definizione matematica di un grafo non include il layout geometrico. Di conseguenza, questo layout non è conservato nel modello, ma nella vista. La vista fornisce un meccanismo di notifica e il supporto per l'undo, che permettono di modificare il layout geometrico, indipendentemente dal modello e dalle altre possibili viste. Poiché il gestore dell'undo di Swing non è adatto per questa configurazione, Jgraph provvede con una estensione del meccanismo di default di Swing per la gestione dell'undo attraverso la clase **GraphUndoManager**.

2.2 Caratteristiche

Le modifiche introdotte (a Jtree) per ottenere delle particolari capacità,
possono essere raggruppate in :

- Ereditarietà : l'implementazione di Jtree per il L&F variabile a run-time e per il supporto alla serializzazione, restano invariate in Jgraph.
- Modifiche : L'implementazione esistente dell'editing in-place e del rendering sono state modificate per lavorare con la viste e la cronologia.
- Estensione : La selezione marquee e la capacità di stepping-into-group estendono il modello di selezione di Jtree.
- Miglioramenti : Jgraph è stato migliorato con il datatransfer, gli attributi e la cronologia, che sono standard Swing non usati però in Jtree.
- Implementazione : Il layering, il raggruppamento (groupping), il clonaggio, lo zoom, le porte e la griglia sono nuove caratteristiche, che sono standard, per quanto riguarda le convenzioni architetturali e di codifica.

2.2.1 Ereditarietà

Il L&G scambiabile e la serializzazione sono usati senza modifiche. Come nel casi di Jtree, il delegato UI implementa il L&F corrente, e la serializzazione è basata sull'intefaccia **Serializable**, mentre le classi **XMLEncoder** e **XMLDecoder** servono per la serializzazione a lungo termine.

2.2.2 Modifiche

La proprietà di supportare tipi multipli di celle influenza il rendering e l'editing in-place di Jgraph. A differenza di Jtree, dove il renderer e l'editor, per l'unico tipo di nodo, è referenziato dall'oggetto albero, in Jgraph gli editor e i renderer sono referenziati dalla vista di cella (cell view), per impedire la ricerca del renderer attraverso il tipo della cella. Il renderer è staticamente referenziato, di modo che, possa essere condiviso tra tutte le istanza di una classe. Inoltre, la vista di cella definisce un handle che permette di estenderne la capacità di editing. L'idea di handle segue da vicino quella usata per il design dell'editing in-place.

2.2.3 Estensioni

Come il modello di selezione di Jtree, quello di Jgraph, permette selezioni di singole celle o di più celle contemporaneamente. Ma il modello di selezione di Jgrah in più permette la selezione step-into dei gruppi. La selezione marquee è inoltre fornita usando un handle di marquee, che è basato sul design del handle di trasferimento.

2.2.4 Miglioramenti

Jgraph permette il trasferimento di celle delle modello, attraverso una descrizione delle strutture di grafo e di gruppo e del loro layout geometrico, o usando il Drag&Drop (DnD) o attraverso gli appunti (clipboard).

Basandosi sulle idee dei componenti di testo di Swing, Jgraph fornisce mappe per descrivere e modificare gli attributi delle celle. Queste mappe incapsulano lo stato della cella e possono essere accedute in una maniera type-safe usando la classe **GraphConstants**.

Jgraph fornisce inoltre la cronologia dei comandi o più semplicemente cronologia, che è l'abilità di fare undo/redo di una modifica precedente. Il design segue quello dei componenti di testo Swing. Sebbene sia necessario un gestore del maccanismo di undo/redo per la presenza di potenziali viste multiple.

2.2.4 Implementazione

Vi sono due gruppi di caratteristiche nell'implementazione :

- Caratteristiche che influenzano solo la classe **Jgraph**
- Caratteristiche che richiedono nuove classi

Il primo gruppo consiste nel *cloning, zoom e griglia* che sono implementate con metodi della classe principale **Jgraph.** Il resto del framework non offre classi e metodi per implementare queste caratteristiche, che dipendono quindi interamente da **Jgraph**.

Il secondo gruppo consiste nel *layering, handles, grouping e ports,* che non sono usate da nessuna altra parte in Swing e perciò richiedono nuove classi e metodi. Particolare attenzione, si deve avere, per basare queste nuove caratteristiche su funzionalità di Swing già esistenti e nel renderle analoghe, per ciò che riguarda il design e l'implementazione. Per esempio, la caratteristica degli handles segue da vicino il design e l'implementazione dell'editing in-place di Swing, di modo che sia facile per il programmatore adottare questa nuova caratteristiche in base alla sua precedente comprensione del meccanismo di editing in-place.

Poiché tali caratteristiche sono nuove, ecco qualche prima spiegazione :

- Layering : poiché le celle possono sovrapporsi, l'ordine con cui esse sono restituite è significativo. L'ordine viene spesso chiamato *layering* e può essere cambiato usando i metodi **toBack** e **toFront** dell'oggetto **GraphView**. Il Layering è parte della vista (view) e viene spiegato nella sezione di questo documento dedicata alla vista del grafo.
- Handles : gli handles, come gli editor, sono oggetti che sono usati per modificare la visualizzazione di un cella. A differenza dell'editing in-place, che usa un componente di testo per modificare il valore di una cella, gli handles usano altre maniere per fornire all'utente un feedback visivo di come il grafo apparirà in seguito ad una esecuzione di una modifica che abbia avuto successo (live-preview). Gli handles e l'editng in-place sono spiegati nella parte sul controllo, poiché è il delegato UI che fornisce queste funzionalità.
- Gruppi e Porte : porte e gruppi sono correlati poiché le porte sono implementate sul top della struttura di gruppo, nel modello del grafo. Il raggruppamento (grouping) è pertanto spiegato nella parte sul modello del grafo, in questo documento.

2.3 Jgraph MVC

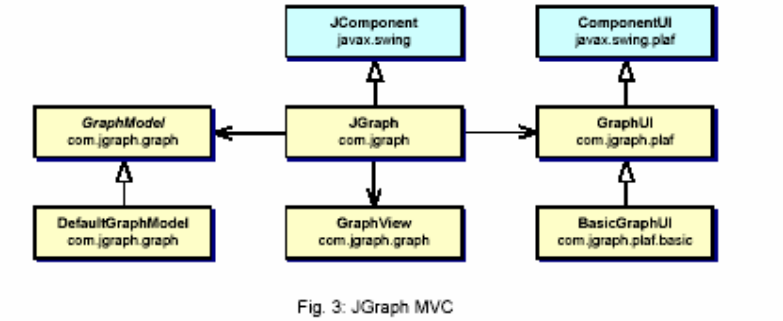

Fig. 3: JGraph MVC

Jgraph estende **Jcomponent** e ha un riferimento al suo **GraphUI**. **Jgraph** ha un riferimento a **GraphModel** e uno a **GraphView** e istanzia **BasicGraphUI,** che estende **GraphUI**, che a sua volta, estende **ComponentUI.**

La struttura di base del componente, ossia l'architettura Swing MVC, è ereditata da Jtree. Sebbene, Jgraph ha un riferimento in più ad una vista del grafo, che non è di solito usata nel MVC di Swing. La vista del grafo è analoga alla vista della root nei componenti di testo Swing, ma essa non referenzia il delegato UI. Invece, essa è referenziata dall'oggetto **Jgraph,** così che esso preservi lo stato, quando il L&F viene cambiato. (L'appendice spiega più in dettaglio MVC, MVC in Swing, e come esso venga applicato a Jtree e a Jgraph).

2.4 Avvio

Quando si lavora con gli attributi (vedi 2.5), la sequenza di avvio è significativa. Il fatto che il modello di default non conserva gli attributi (diremo che non è un "attribute store") deve essere tenuto in considerazione quando si inseriscono le celle, poiché gli attributi di tali celle sono inviate alle viste di cella collegate. Se non vi sono viste collegate, gli attributi vengono ignorati.
Nel caso che una vista sia aggiunta in seguito, la vista usa valori di dafault per la posizione e la grandezza delle celle, il che risulta in celle locate alla stessa posizione e con la stessa grandezza.

Perciò, quando si crea un grafo con un modello customizzato, prima, una istanza di classe **Jgraph** deve essere creata, usando il modello come argomento al costruttore, e solo dopo, le celle devono essere inserite nel modello e non vice versa. Durante la costruzione della istanza di **Jgraph,** una vista viene automaticamente collegata al modello.

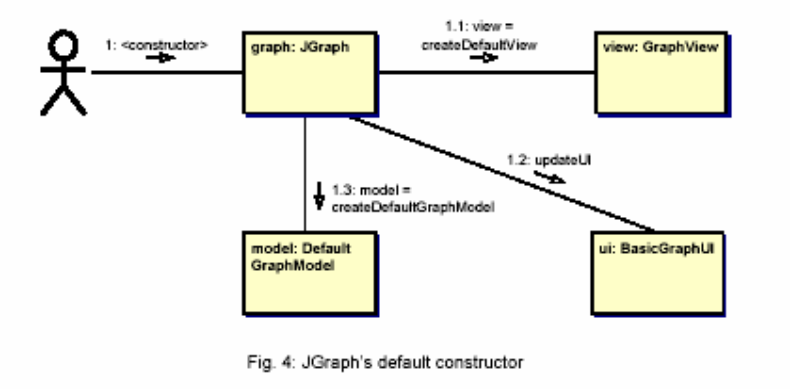

Fig. 4: JGraph's default constructor

La stessa cosa vale quando si setta il modello di un oggetto **Jgraph**, in tal caso infatti la vista viene notificata, e mantiene un riferimento al nuovo modello. In ogni modo, poiché il modello non è un attribute store, la vista userà i valori di default, come nel caso dove sia registrata col modello *dopo* la chiamata al metodo **insert**.

Quanto detto non vale nel caso in cui il modello è un attribute store, nel qual caso, tutti gli attributi sono conservati nel modello invece che nella vista, rendendo queste problematiche di temporizzazione irrilevanti.

2.5 Gli Attributi

Gli attributi di Jgraph sono solo concettualmente basati su quelli di Swing, alcuni aspetti dinamici e la classe che accede a questi attributi sono differenti da Swing e devono perciò essere spiegati.

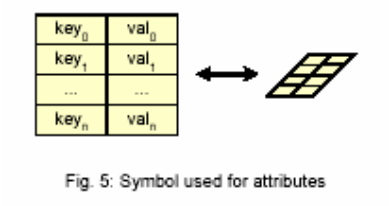

Fig. 5: Symbol used for attributes

Gli attributi sono imlementati usando mappe, da chiavi (key) a valori, e possono essere acceduti usando la classe **GraphConstants.**

2.5.1 GraphConstants

La classe **GraphConstants** viene usata per creare mappe di attributi e per accedere ai valori di queste mappe in una modalità type-safe. Affianco alla capacità di creare e accedere alle mappe, questa classe fornisce un metodo per clonare una mappa e un metodo per applicare una modifica di più di un valore ad una mappa target.

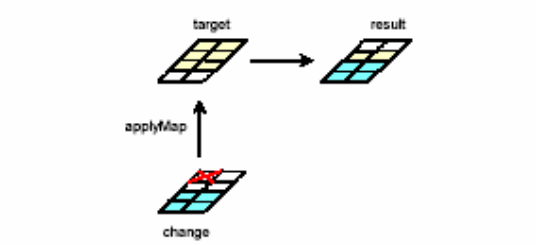

Fig. 6: Changing attributes with the GraphConstants class

Il metodo **applyMap** combina entry (coppie chiave-valore) comuni delle mappe target e source sovrascrivendo quelli della mappa target. Gli entry che sono presenti solo nella mappa change sono inseriti nella target, e quei valori che esistono solo nella target sono lasciati immutati.

Per rimuovere gli entry della mappa target, il metodo **setRemoveAttributes** è usato, fornendo le key che dovrebbero essere rimosse come argomento. Le chiavi sono conservate insieme ai valori nella mappa change e sono gestite dal metodo **applyMap.** Se la mappa di change rimpiazza completamente la mappa target, allora il metodo **setRemoveAll** dovrebbe essere usato sulla mappa change, per indicare che tutte le key della mappa target andrebbero rimosse.

Gli attributi possono essere usati e nel modello e nella vista. In ambo i casi, le mappe degli attributi sono create e accedute usando la classe **GraphConstant**. La relazione tra gli attributi di una cella e quelli della vista di cella corrispondente è come segue :

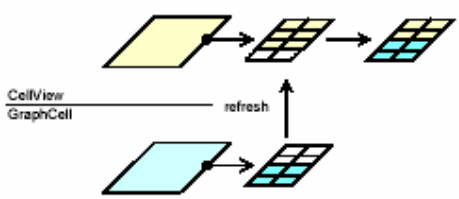

Fig. 7: Relation between a cell's and its view's attributes

Un attributo di cella ha la precedenza su quello di vista omonimo, di modo che la cella possa forzare il valore di un attributo di vista per una specifica key. Ciò vuol dire che, se una vista di cella specifica un valore per una key che è anche specificata dalla cella, allora il valore di cella è usato invece di quello di vista. (Due attributi sono uguali se il metodo **equals** sulle loro rispettive key ritorna true).

In altre parole, gli attributi blu hanno la precedenza su quelli gialli, e se un attributo non è presente nella mappa gialla, allora vi verrà inserito. I valori gialli che non esistono come blu sono lasciati immutati. (Poiché questo meccanismo è basato sul metodo **applyMap**, il suo comportamento è esattamente lo stesso.)

La classe **GraphConstants** non distingue tra gli attributi per le celle o per le viste di celle, poiché sono basate sulla stessa sottostante struttura. A differenza delle celle, che accettano tutti gli attributi, le viste di cella eseguono dei test addizionali su ciascuna key. Il renderer delle viste viene usato per determinare se la key è supportata , se non lo è, l'entry è rimosso dalla corrispondente mappa degli attributi di vista, per eliminare ridondanza.
(I metodi per settare gli attributi sono tutti basati su **applyMap**.)

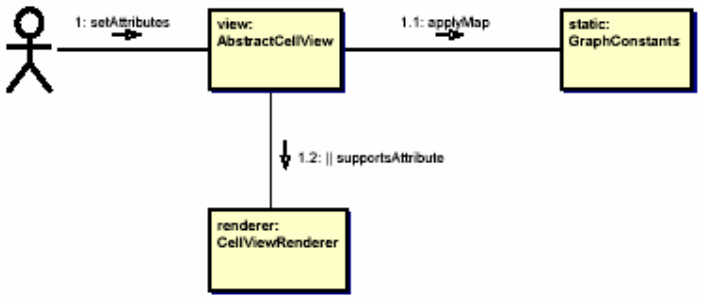

Fig. 8: Implicit use of the GraphConstants class

Nella implementazione di default di Jgraph, la UI cambia gli attributi di vista in seguito alle interazioni utente. Forzando il metodo **isAttributeStore**, il modello può ricevere il controllo. Se il metodo ritorna true, allora gli attributi di cella saranno modificati invece di quelli vista, con il risultato che tutte le viste di grafo attaccate saranno aggiornate. Nel caso di default, solo la vista locale è aggiornata (sebbene la modifica riguardi il modello).

Questo perché tutte le viste sono aggiornate in seguito ad un cambio del modello, attraverso il meccanismo di notifica, e gli attributi di cella sono usati in tutte le viste. Una eccezione è per l'attributo **value**, che è in sincronia con lo **user object** della cella. L' attributo **value** è memorizzato comunque nel modello, indipendentemente se esso sia o no un attribute store.

2.5.2 L'Attributo Value

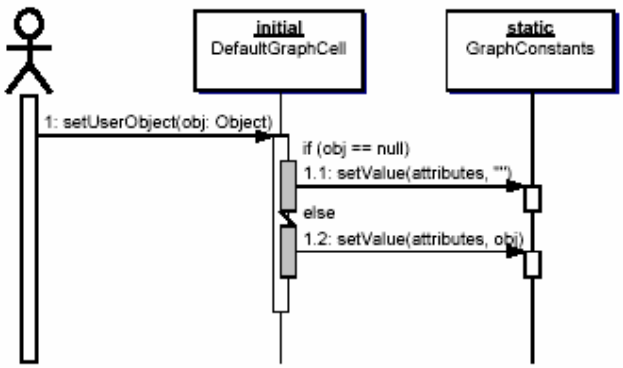

Fig. 9: Synchronization upon change of a cell's user object

Il metodo **setUserObject** della classe **DefaultMutableTreeNode** è forzato dalla classe **DefaultGraphCell** di modo che lo **user object** passato sia conservato sotto la key **value** nella mappa di attributi della cella. Vice versa, il metodo **setAttributes** sovrascrive il precedente **user object** se un nuovo oggetto per la key **value** è specificato. Perciò, l'attributo **value** punta allo **user object** e vice versa.

L'attributo **value** è usato nel contesto della cronologia e del'editing in-place. Introducendo l'attributo **value,** lo stato completo della cella può essere rappresentato dai suoi attributi; lo **user object** non richiede una gestione speciale. Una modifica allo stato (e ugualmente allo **user object**) può essere annullata (undo) usando il metodo **applyMap,** fornendo il precedente e il corrente stato come argomenti.

Per default, l'editing in-place rimpiazza lo **user object** con una nuova stringa, il che non sempre è desiderabile. Perciò, lo **user object** può implementare l'interfaccia **ValueChangeHandler**, che cambia il comportamento di default.

2.5.3 L'Interfaccia ValueChangeHandeler

L'interfaccia **ValueChangeHandler**, che è una interfaccia annidata della classe **DefaultGraphCell**, può essere usata per evitare che lo **user object** sia sovrascritto durante l'editing in-place. Se lo **user object** implementa questa interfaccia, allora **DefaultGraphCell** lo informa della modifica, ed egli è responsabile di memorizzare il nuovo e restituire il vecchio **value**. Lo **user object** riflette il cambio attraverso il metodo **toString**.
(il metodo **convertValueToString** è usato per convertire una cella in una stringa).

2.5.4 Attributi e Campi

Le nuove proprietà dovrebbero essere implementate come attributi o come campi ? Nel primo caso, il clonaggio della cella e il meccanismo di undo sono già pronti, ma l'attributo deve essere acceduto attraverso una hash-table. Nel secondo caso, il clonaggio della cella richiede una gestione speciale, ma la variabile può essere acceduta come un campo, il che potrebbe essere richiesto per implementare l'ereditarietà. La combinazione dei due porta ad un incremento di ridondanza e complessità.

Fondamentalmente, gli attributi dovrebbero essere usati solo per il rendering, anche se non ci sono restrizioni per memorizzare attributi custom. Poiché i soli attributi supportati sono propagati alla vista, un tale attributo custom non aggiunge ridondanza per quanto riguarda gli attributi della vista. (L'attributo **value** fa eccezione essendo un campo supportato dal renderer).

Invece di estendere la classe **DefaultGraphCell** o di usare gli attributi per memorizzare informazioni custom, la classe, che eredita da **DefaulMutableTreeNode** fornisce anche un altro meccanismo per memorizzare dati custom, attraverso lo **user object**. Lo **user object** è di tipo **Object** e perciò fornisce un modo per associare qualsiasi oggetto con una cella.

Lo **user object** può essere in una gerarchia di classe arbitraria, per esempio potrebbe estendere un hash-table. Poiché il metodo **toString** dello **user object** è usato per fornire l'etichetta, questo metodo dovrà probabilmente essere forzato.

2.6 Clonaggio

Una nuova caratteristica di Jgraph è la possibilità di clonare la celle automaticamente. Questa caratteristica è costruita attraverso l'implementazione di default degli appunti e attraverso gli handle di cella, ed è basata sul metodo **clone** di ogni oggetto e sul metodo **cloneCells** di Jgraph. Questa caratteristica può essere disabilitata usando il metodo **setClonable** nell'oggetto **Jgraph**. (Disabilitando questa caratteristica, un Control-Drag sarà interpretato come un move normale).

Il processo di clonaggio è diviso in una fase locale ed una globale : nella locale, ciascuna cella è clonata usando il suo metodo **clone**, ritornando un oggetto che non referenzia altre celle. La cella e il suo clone sono memorizzati in una hash-table, usando la cella come key e il clone come valore.

Nella fase globale, tutte le celle referenziate dalla cella originale del clone sono rimpiazzate da referenze(riferimenti) ai loro corrispondenti cloni (usando la hash-table). Perciò, nel processo di clonaggio delle celle, prima tutte le celle sono clonate usando **clone**, poi tutti i riferimenti a celle sono consistentemente rimpiazzati con riferimenti ai rispettivi cloni.

2.7 Zoom e Griglia

Jgraph usa la classe **Graphics2D** per implementare lo zoom. Il framework si affida a metodi per scalare un punto o un rettangolo da coordinate modello a coordinate schermo e viceversa. In tal modo il codice client è indipendente dall'attuale fattore di zoom.

Poiché lo zoom di Jgraph è implementato sul top della classe **Graphics2D**, il disegno sull'oggetto grafico usa coordinate non scalate (lo scaling attuale è ottenuto dall'oggetto grafico stesso).Per tale motivo, Jgraph ritorna sempre e si aspetta coordinate non scalate.

Per esempio, quando si implementa un listener del mouse per rispondere ai click del mouse, il punto dell'evento dovrà essere sottoscalato alle coordinate di modello, usando il metodo **fromScreen,** per trovare la cella corretta col metodo **getFirstCellForLocation.**

Dall'altra parte, il punto originale è di solito usato nel contesto dei componenti, per esempio per mostrare un menù di pop-up, sotto il click del mouse. Ci si assicuri di clonare il point che sarà scalato, poiché il metodo **fromScreen** modifica l'oggetto passato senza farne una copia. Per scalare dal modello allo schermo, per esempio per trovare la posizione di un vertice sul componente, il metodo **toScreen** è usato.

Per supportare la griglia, ciascun punto che viene usato dal grafo deve essere applicato alla griglia usando il metodo **snap**. Come nel caso dello zoom, il metodo **snap** modifica l'argomento, invece di clonarlo e modificare il clone (punto, passato per riferimento non per valore). Ciò perché l'instanziazione in Java è costosa, e non sempre si richiede di non modificare l'argomento. Perciò, il clonaggio dell'argomento è lasciato al codice client.

Jgraph fornisce due *proprietà bound* che appartendono alla griglia : una per renderla visibile, l'altra per abilitarla. Perciò, la griglia può essere visibile, ma ancora disabilitata, o abilitata ma non visibile.

3 Model

Il modello fornisce i dati per il grafo, che consistono in celle, che possono essere vertici, spigoli e porte, e in informazioni di connessione, che sono definite usando le porte, che fanno da sorgente e destinazione per uno spigolo. Queste informazioni di connessione sono chiamate *stuttura del grafo*. (il layout geometrico non è considerato parte della struttura del grafo).

3.1 Raggruppamento

Il modello fornisce l'accesso ad un'altra struttura, chiamata la *struttura di gruppo.* La struttura di gruppo consente alle celle del grafo di essere inserite in una gerarchia parte-tutto, cioè permette di creare nuove celle da quelle esistenti.

Il seguente grafo contiene un tale gruppo, che contiene il vertice A e B e lo spigolo 1. Il vertice A contiene la porta a come figlia, e il vertice B contiene le porte b0 e b1 come figlie

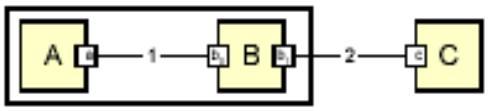

Fig. 10: A graph with a group

Poiché la struttura di gruppo aggiunge una dimensione al modello del grafo, la figura di sopra non permette di visualizzare entrambe le strutture, anche se il solo grafo è pienamente definito.
(La struttura di gruppo in questa figura è disegnata usando un rettangolo tratteggiato, ma in realtà, essa non è tipicamente visibile.)

Per illustrare la sottostante struttura di gruppo, un'immagine tridimensionale è usata, che rappresenta la struttura del grafo nel piano X-Y, e i piani della struttura di gruppo shiftati lungo l'asse Z.

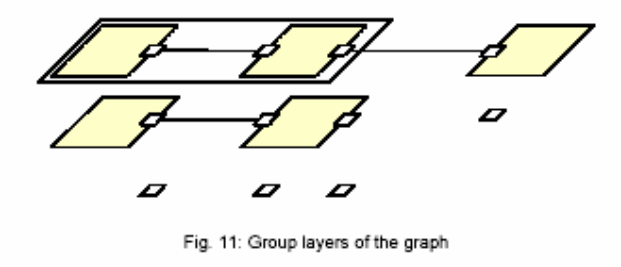

Fig. 11: Group layers of the graph

Il diagramma dei piani di gruppo mostra la modalità di raggruppamento del grafo.
I rettangoli tratteggiati sono usati per illustrare la visibilità delle celle sul piano più alto;
le istanze attuali delle celle "vivono" nel piano più basso. Nota che le porte sono parte della struttura di gruppo e appaiono come figlie dei loro vertici.

Per illustrare la struttura di gruppo da sola, un altro diagramma è usato. In questo diagramma, i gruppi appaiono come alberi, con le radici memorizzate in liste collegate. Gli elementi sono disegnati come cerchi, per sottolineare lo scopo diverso e il contenuto del diagramma. La cella vuota alla radice di A, 1 e B è un gruppo che non ha etichetta. Non è possibile dedurre il grafo originale dal diagramma della struttura di gruppo.

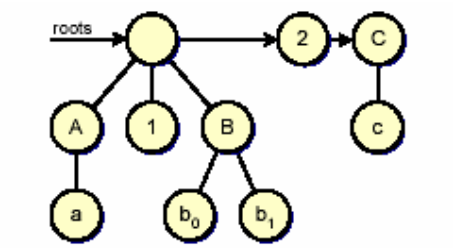

Fig. 12: Group structure of the graph

La struttura di gruppo può essere pensata come una foresta di alberi, dove le radici di questi alberi posso essere ottenute usando l'interfaccia **GraphModel**. Una volta che queste radici sono state ottenute dal modello, i loro discendenti possono essere ottenuti usando i rispettivi metodi dell'interfaccia **GraphModel**.

Logicamente, la struttura di gruppo può essere vista come una estensione dell'interfaccia **TreeModel.** Sebbene, una tale estensione è tecnicamente impossibile poiché l'interfaccia **TreeModel** fa riferimento all'interfaccia **TreeModelListeners**, che non può essere usata nel contesto dei grafi.

Un importante fatto che dovrebbe essere tenuto in mente è che la sorgente e la destinazione di uno spigolo implementano l'interfaccia **Port**, che è usata come una indirezione per fornire punti di connessione multipla per uno stesso vertice. La relazione tra la porta e il vertice è implementata nella parte più alta della struttura di gruppo, essendo il vertice il genitore della porta. (Le porte sono qualcosa di artificioso, e da un punto di vista strettamente matematico, non appartengono alla definizione di un grafo.)

Per riassumere, il modello del grafo non solo fornisce l'accesso alla struttura del grafo, ma da accesso anche ad una struttura indipendente, chiamata la struttura di gruppo, che permette alle celle di essere inserite in gerarchie parte-tutto. La struttura di gruppo può essere vista come un insieme di alberi, dove le radici degli alberi rappresentano i genitori dei gruppi (se un nodo virtuale venisse aggiunto come genitore di queste radici, allora il modello stesso sarebbe un albero).
La struttura di gruppo invece fornisce i metodi per ottenere le porte sorgenti e le destinazioni di uno spigolo e per restituire gli spigoli che sono connessi alla porta o ad un array di celle (porte).

3.2 Celle

Le celle del grafo sono l'ingediente chiave di Jgraph che fornisce l'implementazione di default delle celle più comuni : vertici, spigoli e porte. Il vertice è considerato come il caso di default che non necessita di un'interfaccia ed usa l'implementazione di default della superclasse comune. Le porte non devono essere trattate come figli speciali, di modo che le loro relazioni con il vertice possano essere implementate sul top della struttura di gruppo, fornita dall'interfaccia **GraphModel**.

Questa interfaccia contiene solo la parte *modello* del framework, la *vista* ha una diversa rappresentazione interna del grafo. Nella struttura della vista del grafo, figli e porte sono trattate come entità differenti e sono conservate in liste differenti. Una delle ragioni per questo è che le porte (se visibili) appaiono sempre più in alto sul grafo, indipendentemente dal loro ordine di visualizzazione e la seconda ragione risiede in un incremento di performance del framework.

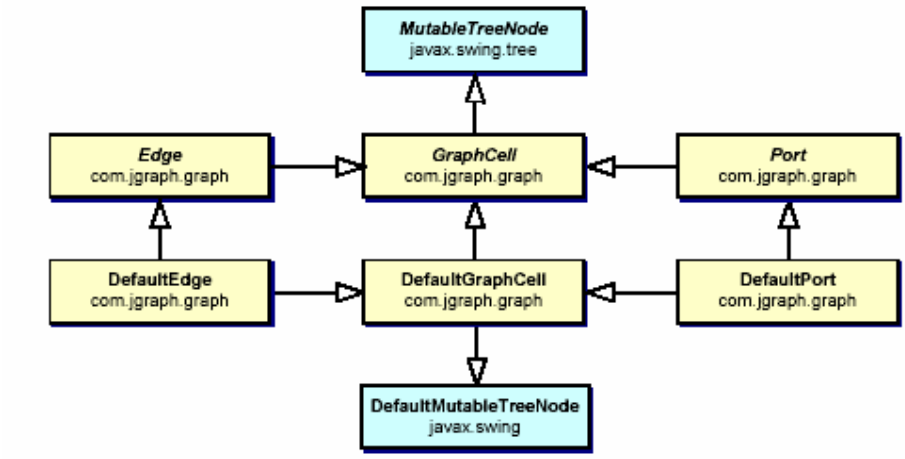

Fig. 13: GraphCell interface hierarchy and default implementations

DefaultPort e **DefaultEdge** estendono **DefaultGraphCell, Port** e **Edge** rispettivamente, che dalla sua estende **GraphCell**.
DefaultMutableTreeNode ha un riferimento ad un **TreeNode** che rappresenta il genitore e ad un array di **TreeNode,** che sono i figli.
DefaulEdge mantiene un riferimento a due **Object**, la sorgente e la destinazione, che di solito estendono **Port**. **DefaultPort** ha un riferimento a **Port** che rappresenta l'*anchor* e instanzia un insieme che contiene gli spigoli collegati.

3.2.1 Gerarchia delle Interfacce di Cella

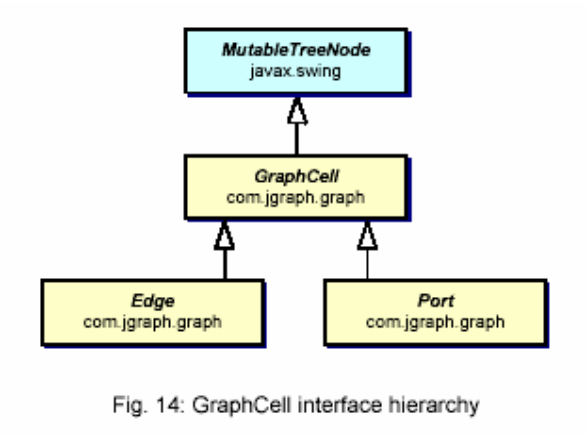

Fig. 14: GraphCell interface hierarchy

L'interfaccia **GraphCell** è una estensione dell'interfaccia Swing **MutableTreeNode**, che fornisce i metodi per **DefaultGraphModel** per memorizzare le sue strutture interne di grafo e di gruppo.
L'interfaccia **GraphCell** stessa fornisce metodi per aver accesso agli attributi della cella, mentre i metodi per accedere ai figli ed ai genitori sono ereditati dall'interfaccia **MutableTreeNode**.

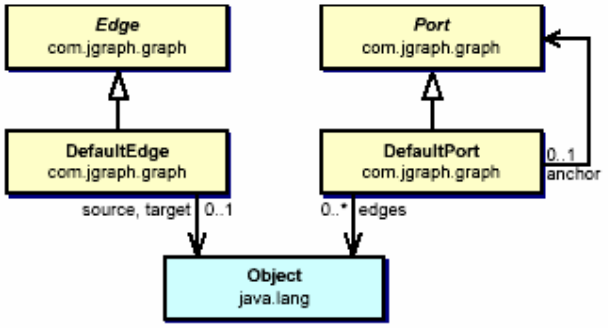

Fig. 15: The graph structure is stored in the edges and ports

L'interfaccia **Edge** fornisce i metodi per accedere alle porte sorgente e destinazione e l'interfaccia **Port** accede a tutti gli spigoli attaccati ad una specifica porta.
Perciò la struttura di grafo utilizza solo le interfacce **Port** e **Edge**.
Come analogia con il componente text, l'interfaccia **GraphCell** è l'analoga dell'interfaccia **Element** [15], mentre **CellView** è l'analogo Jgraph per l'interfaccia **View** del componente text[16]

3.2.2 L'implementazione di Default di GraphCell

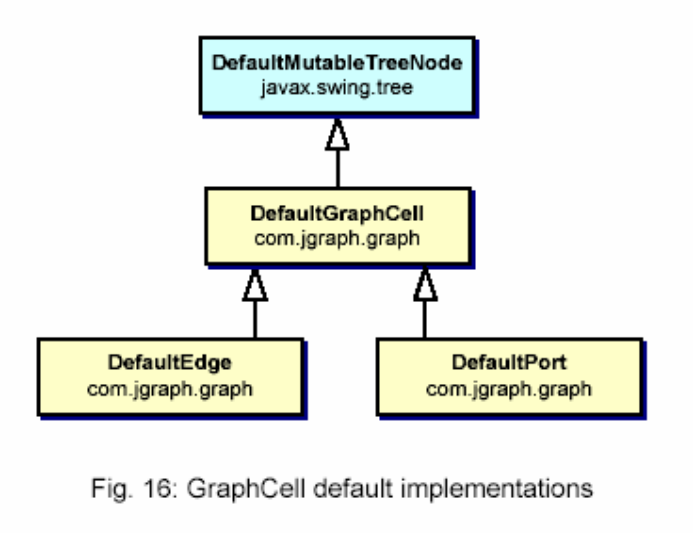

Fig. 16: GraphCell default implementations

La classe astratta di base, chiamata **DefaultGraphCell**, fornisce le funzionalità comuni a tutte le celle del grafo per :

- Clonare celle e i loro figli
- Gestire gli attributi di cella
- Sincronizzare l'attributo **value** con lo **user object**

La classe **DefaultGraphCell**, che è la superclasse comune di tutte le celle è una estensione di **DefaultMutableTreeNode** e come tale tutte le celle di Jgraph possono essere inserite dentro gruppi.

L'implementazione di default non fa distinzioni tra celle usate come gruppo o no, e non fa restrizioni su ciò che un genitore o un figlio può accettare. Il programmatore deve assicurarsi che si possano creare i soli gruppi che abbiano un senso nel contesto del applicativo.
Tipicamente, i vertici sono usati cone gruppi. Un vertice che è anche un gruppo è invisibile e appare solo come una selezione intorno ai suoi figli quando il gruppo stesso è selezionato.

Cosa dire di un **DefaultEdge** usato come gruppo? Il renderer disegnerà uno spigolo tra le porte sorgenti e destinazione specificate, se disponibili.Un possibile uso è la visualizzazione di lunghi percorsi tra due vertici, dove i salti nel percorso non sono importanti.

Bisogna fare attenzione alla compatibilità con il renderer in questo caso, poiché il renderer potrebbe aspettarsi certi attributi per funzionare correttamente. Pertanto, se una cella invece di un vertice è usata come gruppo, essa deve anche contenere gli attributi che sono richiesti dai rispettivi renderer. Le porte possono essere usate come gruppi, il che è ancora più "patologico" come caso. Per ulteriori spiegazioni sulla sincronizzazione tra l'attributo **value** e lo **user object**, si legga il capitolo sugli attributi.

3.3 Il Modello del Grafo

3.3.1 L'Interfaccia GraphModel

```
public interface GraphModel {
    // Attributes
    Map getAttributes(Object node);
    boolean isAttributeStore();
    // Roots
    int getRootCount();
    Object getRootAt(int index);
    int getIndexOfRoot(Object root);
    boolean contains(Object node);
    // Layering
    void toBack(Object[] cells);
    void toFront(Object[] cells);
    boolean isOrdered();
    // Graph structure
    Object getSource(Object edge);
    Object getTarget(Object edge);
    Iterator edges(Object port);
    boolean acceptsSource(Object edge, Object port);
    boolean acceptsTarget(Object edge, Object port);
    // Group structure
    Object getParent(Object child);
    int getIndexOfChild(Object parent, Object child);
    Object getChild(Object parent, int index);
    int getChildCount(Object parent);
    boolean isLeaf(Object node);
    // Change support
    void insert(Object[] roots, ConnectionSet cs, ParentMap pm, Map attributeMap);
    void remove(Object[] roots);
    void edit(ConnectionSet cs, Map nestedMap, ParentMap pm, UndoableEdit[] e);
    // Listeners
    void addGraphModelListener(GraphModelListener l);
    void removeGraphModelListener(GraphModelListener l);
    void addUndoableEditListener(UndoableEditListener listener);
    void removeUndoableEditListener(UndoableEditListener listener);
}
```

L'interfaccia **GraphModel** definisce le specifiche per quegli oggetti che possono servire come sorgente di dati per il grafo. I metodi definiti in tale interfaccia forniscono accesso alle due strutture indipendenti :

- La struttura di grafo
- La struttura di gruppo

La struttura di grafo segue la definizione matematica di un grafo, dove gli spigoli hanno una sorgente ed una destinazione e i vertici hanno un insieme di di spigoli connessi.
La connessione tra vertici e spigoli è rappresentata da una entità speciale nota come porta. In un certo senso, si può affermare, che le porte sono figlie delle celle e appartengono alla struttura di gruppo. Sebbene sono anche usate per descrivere le relazioni intercorrenti tra spigoli e vertici, che le rende anche parte della struttura del grafo.

In più, **GraphModel** definisce i metodi per gestire e registrare due tipi differenti di listener : **UndoableEditListener** e **GraphModelListener**. I due sono in qualche modo collegati, poiché ciascuna notifica del listener di undo è accompagnata da una notifica del listener del modello.

Il caso inverso non sussiste : infatti se su di un cambiamento viene eseguito un undo o redo, i listener del modello sono notificati per aggiornare la vista e ridisegnare lo schermo, ma la cronologia dei comandi non deve essere aggiornata, poiché essa già contiene questo specifico cambiamento.

Un'altra caratteristica importante di **GraphModel** è la sua abilità di decidere quali connessioni permettere. Per questo scopo i metodi **acceptsSource** e **acceptsTarget** sono invocati dalla UI del grafo prima che uno spigolo è connesso o disconnesso da una porta, ricevendo come argomenti lo spigolo e la porta.

Forzando questi metodi, un modello customizzato può restituire true o false per una coppia spigolo-porta specifica, per indicare se questa connessione è valida, ossia se può essere effettuata dalla UI del grafo. (Questi metodi sono anche invocati quando gli spigoli sono disconnessi, in tal caso la porta è null)

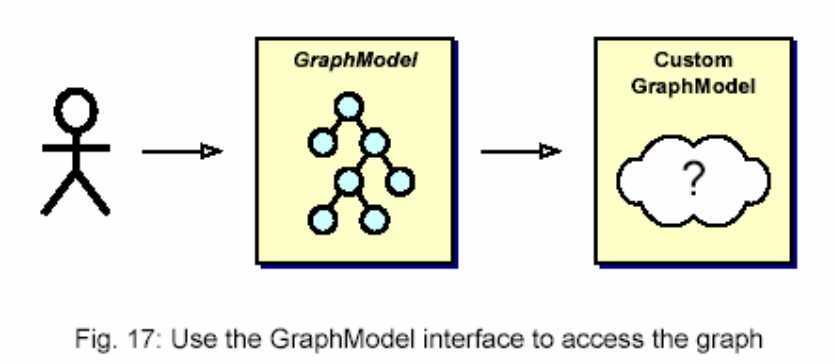

Fig. 17: Use the GraphModel interface to access the graph

E' una pratica generale, quella di usare **GraphModel** per accedere alle strutture di grafo e gruppo, invece di usare le interfacce **GraphCell** e **TreeNode** che forniscono le stesse funzionalità.
La motivazione per questo è che **DefaultGraphModel** di per se dipende da queste interfacce per implementare i suoi metodi, di modo che in future implementazioni, un modello possa contenere oggetti che non implementano l'interfaccia **GraphCell**, permettendo in tal modo al modello di conservare i dati in un modo differente. Se si accede al grafo attraverso **GraphModel**, il codice resta indipendente dalla struttura interna del modello, così che l'implementazione possa essere scambiata senza dover cambiare il codice client.

3.3.2 L'implementazione di Default di GraphModel

Jgraph fornisce una implementazione di default dell'interfaccia **GraphModel** con la classe **DefaultGraphModel**, che usa **GraphCell**, le sue superclassi e l'interfaccia **MutableTreeNode** per accedere alle strutture gruppo e grafo. Ciò significa che il modello conserva i dati nelle celle, cioè, le celle fanno i modelli delle strutture di gruppo e grafo.

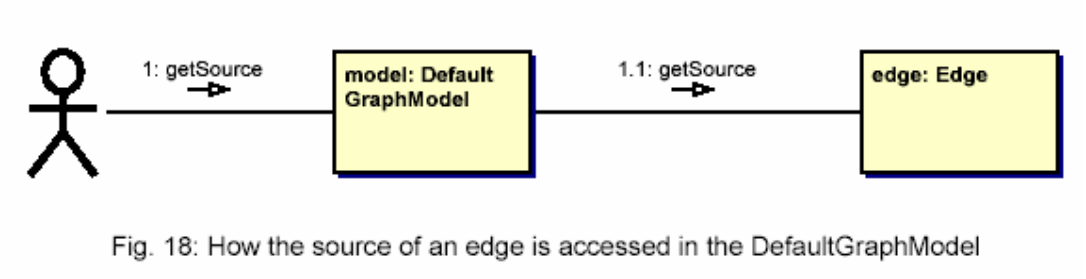

Fig. 18: How the source of an edge is accessed in the DefaultGraphModel

In analogia con **DefaultMutableTreeModel,** che conserva i dati nei nodi, il modello **DefaultGraphModel** di Jgraph mentiene i dati nelle celle. Ciò è efficiente, ma se le celle devono essere usate in modelli multipli, con relazioni diverse in ogni modello, allora **DefaultGraphModel** non può essere utilizzato. Questo accade perché tutte le relazioni vengono conservate nelle celle, rendendo impossibile determinare l'insieme di relazioni che appartengono ad un determinato modello. (Lo stesso vale per **DefaultMutableTreeNode**).

Consideriamo un esempio di grafo che contiene due vertici A e B e uno spigolo. Ciascuno dei due vertici ha una porta come solo figlio, la porta *a* per A e *b* per B, come in figura :

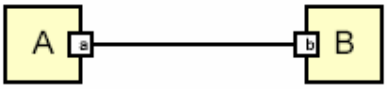

Fig. 19: A graph with two vertices and ports, and one edge in between

La figura più in basso, mostra come **DefaultGraphModel** rappresenta il grafo.
Le celle radici, ossia quelle che hanno come genitore null, sono conservate in una lista e possono essere ottenute usando i metodi **getRootCount** e **GetRootAt**. I figli non sono contenuti in tale lista di roots, ma si accede ad essi per tramite dei metodi **getChildCount** e **getChildAt**, fornendo come argomenti i genitori e, se applicabile,l'indice del figlio.

L’ordine imposto da questi metodi è usato come layering di default dalla vista del grafo. Il layering può essere cambiato individualmente per ciascuna vista, rendendo possibile ad ogni vista di fornire un differente layering.

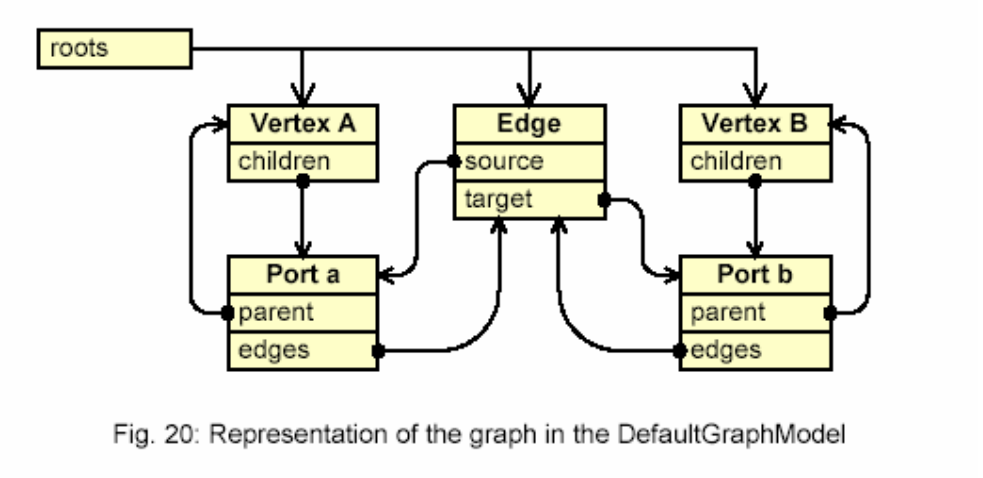

Fig. 20: Representation of the graph in the DefaultGraphModel

La figura sopra, mostra che **DefaultGraphModel** contiene una certa quantità di ridondanza per ciò che concerne l’immagazinamento delle strutture di gruppo e grafo. L’insieme di spigoli connessi ad una porta sono deducibili dal campo source e target dello spigolo, ma ciò richiede un attraversamento del grafo, che esamini le sorgenti e le destinazioni di tutti gli spigoli.

Vice versa, ottenere la sorgente e la destinazione di uno spigolo è computazionamente ancora più costoso, in quanto richiede un pieno attraversamento del grafo e un attraversamento degli insiemi di spigoli di ogni porta. A causa di queste costose necessità di percorrenza del grafo, una certa ridondanza viene introdotta come una cache per migliorare le performance di accesso al grafo.

Il **DefaultGraphModel** implementa i metodi **acceptsSource** e **acceptsTarget** che ritornino true per ogni argomento ed il metodo **isAttributeStore** che restiuisce false, il che indica che il **DefaultGraphModel** non è un immagazzinatore di attributi. Ma estendendo **DefaultGraphModel**, questi metodi possono essere forzati a restituire qualcosa di più significativo per quanto riguarda l’applicazione che ne fa uso.

3.4 Manipolazioni del Modello

I cambiamenti all'implementazione del modello sono atomici, e possono essere conservati in una cronologia dei comandi per un successivo undo/redo. Le manipolazioni possibili sono :

- Inserzione
- Rimozione
- Cambio

Cambiamenti compositi, ossia tali da combinare due o più di questi, non sono forniti dall'interfaccia **GraphModel**, sebbene, tali cambiamenti a volte sono utilizzati internamente, dalla implementazione di default **DefaultGraphModel.**

Cambiamenti al layering delle celle potrebbero essere implementati nel modello, ma nella implementazione di default, la classe **GraphView** fornisce questa funzionalità.

Per inserire o cambiare le celle, un certo numero di classi è usato come argomenti ai rispettivi metodi :

- Mappe identate, da celle ad attributi che permettono di definire attributi nuovi o cambiati
- La classe **ConnectionSet** permette alla struttura di grafo di implementare le sue connessioni, permettendo di crearle e di disconnetterle.
- La classe **ParentMap** descrive la struttura di gruppo, e consiste di coppie padre-figlio.

3.4.1 Mappe Annidate

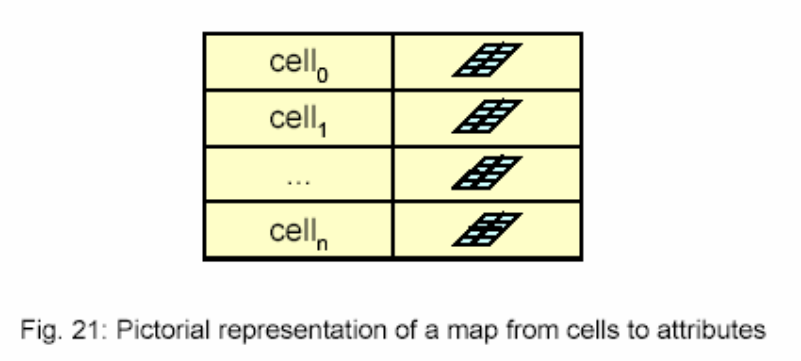

Fig. 21: Pictorial representation of a map from cells to attributes

I metodi **insert** e **edit** permettono di specificare un mappaggio tra celle e attributi**.**
Perciò, non è una mappa, ma una mappa di mappe. Ciò può causare confusione, specialmente quando si cambia una cella, in tal caso la mappa più esterna contiene solo una entità. Per mantenere **GraphModel** semplice, questa è la sola maniera con cui le celle possono essere cambiate, poiché questo è il metodo più generico.

3.4.2 ConnectionSet

La classe **ConnectionSet** si usa per costruire dei cambi alla struttura del grafo, ossia modifica la connessione del grafo. Aggiungendo coppie spigolo-porta, si definiscono le connessioni, da inserire o rimuovere, che verranno fatte. Il nome contiene il termine set perché questo oggetto impone una semantica di tipo set ai dati che contiene.
Internamente, le connessioni sono descritte da istanze di classe **Connection**, che è una classe annidata di **ConnectionSet**. Due connessioni sono uguali se descrivono la sorgente e la destinazione dello stesso spigolo, e le connessioni introdotte sovrascrivono (forzano) quelle esistenti. (Per essere precisi : La classe **Connection** mantiene un riferimento a uno spigolo, una porta e un boolean che indica se la porta è sorgente o destinazione. L'eguaglianza è definita in base ai valori dello spigolo e al boolean.)

Un connection set, che descrive le connessioni esistenti per un array di celle in un dato modello, può essere creato usando un *metodo factory*, che è un metodo statico di classe che ritorna una istanza di **CollectionSet**. Il risultante connection set può sia descrivere la connessione, o la disconnessione delle celle specificate, in un dato modello, in base al valore del boolean.

3.4.3 ParentMap

La classe **ParentMap** è usata per cambiare la struttura di gruppo. Usando la parent map, coppie figlio-padre sono conservate, che sono in seguito usate per stabilire una relazione padre-figlio tra le celle specificate nel modello. Il termine map è uato poiché questo oggetto è in effetti una mappa da figli a padri. Così come in tutte le mappe, le key sono conservate in un set, così che, per ogni figlio che esiste ci sarà esattamente un padre in una parent map. Come con i connection set, i vecchi ingressi sono forzati dai nuovi se lo stesso figlio è usato due volte.

Internamente, la parent map conta il numero di figli che un padre avrà dopo la sua esecuzione. Un metodo è fornito per accedere a questo contatore, in modo che futuri padri vuoti possono essere marcati per essere rimossi durante la costruzione della transazione. In questo senso, vi sono transazioni che combinano il cambio e la rimozione delle celle, ma questa caratteristica non è disponibile attraverso l'interfaccia **GraphModel**. (E' usata internamente, per rimuovere gruppi che non hanno figli.)
La classe **ParentMap** provvede un costruttore che può essere usato per creare la struttura di gruppo per un array di celle in un dato modello. L'oggetto può essere sia costruito per stabilire queste relazioni sia per rimuoverle.

3.4.4 Inserire celle

Quando si inseriscono celle nel modello, solo la cella più in alto deve essere inserita. Poiché il modello chiede alle celle di restituire i figli, questi sono inseriti implicitamente, inserendo i padri. Perciò, quando si inserisce una gerarchia, solo la cella più in alto deve essere inserita.
(Mentre è possibile inserire i figli di una cella nella lista delle roots del modello, di solito questo dovrebbe essere proibito.)

Per l'inserzione, il modello verifica se esso è un attribute store, e se lo è, assorbe gli attributi memorizzandoli localmente nelle celle. (Se il modelo non è un attribute store, allora gli attributi sono passati alle viste, e non sono usati nel modello.) Un oggetto che descrive l'inserzione è creato, esecuto ed inviato ai listener di undo e di modello usando il metodo **postEdit** del modello

3.4.5 Rimuovere Celle

Quando si rimuovono le celle dal modello, i figli devono essere trattati in un modo speciale. Non facendolo, se i figli non sono rimossi durante la rimozione, l'operazione risultante è un ungroup. Cioè le celle specificate vengono rimosse dalla lista di roots del modello, e la prima generazione di figli è rimossa dalla cella, ed è aggiunta al padre del padre.
Un altro caso particolare è la rimozione di tutti i figli da un gruppo, senza rimuovere la cella gruppo stessa. In tal caso, il modello di default rimuove il gruppo automaticamrente. (Ciò accade anche quando tutti i figli di un gruppo sono spostati verso un altro padre.)

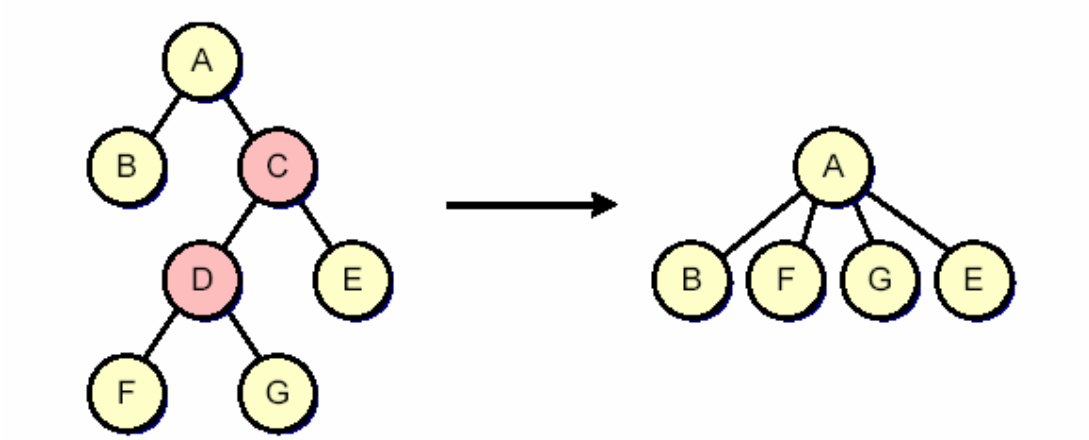

Fig. 22: Group structure before and after the removal of C and D without children

Sulla sinistra , il gruppo A contiene un gruppo C che contiene un gruppo D e una cella E. Il gruppo D contiene le celle F e G. In figura sulla destra, la risultante struttura di gruppo consiste di un gruppo A, che contiene le celle B, E, F e G. I figli delle celle rimosse sono stati aggiunti ai padri del padre , dunque, solo loro diventano roots del modello, se il padre era una cella root esso stesso, ossia, se il suo rispettivo padre è null.

3.4.6 Cambiare le Celle

Quando le celle sono modificate, ci sono quattro argomenti da passare al metodo **edit** : il connection set per cambiare la struttura di grafo, la parent map per cambiare la struttura di gruppo, una mappa di mappe per cambiare gli attributi, e un array di cambiamenti su cui si può eseguire l'undo. Il terzo argomento è una mappa, da celle ad attributi, che sono mappe da key a valori. (L'ultimo argomento è usato internamente per supportare l'undo, da parte della vista.)

Se l'ultimo argomento nella chiamata del metodo **edit** è il solo non null, allora il modello invia i cambi da undoare al listener di undo registrato con il modello. In tutti gli altri casi, il modello crea

un oggetto che descrive la modifica, la esegue e notifica i listener di undo e di modello usando il metodo **postEdit**.

3.5 Modello di Selezione

Logicamente, il modello di selezione di Jgraph è una estensione di quello di Jtree, offrendo tutte le sue funzionalità, come la possibilità di selezionare una o più celle. Di più, il modello di selezione di Jgraph permette lo stepping nei gruppi, come detto prima. Il compito del modello di selezione è di calcolare la corrente selezione e di garantire specifiche proprietà sulla selezione. Per esempio, deve garantire che non sia possibile selezionare una cella e un suo discendente, mai allo stesso momento.

Per ottenere i candidati alla selezione, ossia quelle celle che sono in questo momento selezionabili, l'interfaccia fornisce il metodo **getSelectable.** La capacità di stepping into può anche essere spenta, in tal caso il modello di selezione offre i normali modi di selezione singola e multipla di celle, così come fa Jtree.

3.5.1 Stepping-into per i Gruppi

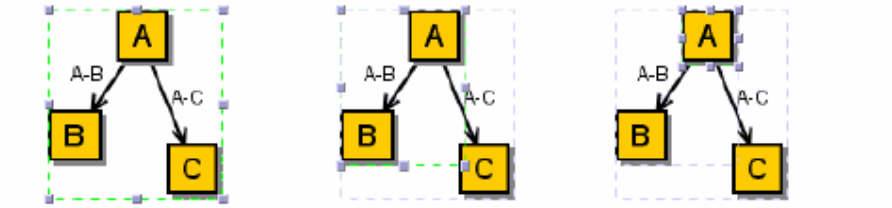

Fig. 23: Stepping-into groups in a UI, from the outermost group to the innermost cell

Per spiegare le caratteristiche di stepping into, è importante sottolineare l'ordine con cui le celle vengono selezionate. Come base per questo ordinamento si usa il layering, che dipendente dalla corrente vista, oppure l'ordine di modello se il modello è un attribute store.

L'ordine in cui le celle sono selezionate è definito dalla sequenza di celle selezionabili, che è induttivamente definito dal seguente, ed è inizialmente uguale alle celle della lista di roots del modello. (Chiaramente, una cella può essere selezionata solo se è selezionabile, quindi se è contenute nella sequenza di celle selezionabili.)

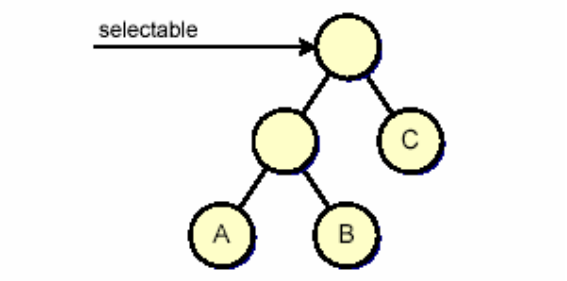

Fig. 24: Initially, the root cells are selectable

Se l'utente clicca sul grafo, la sequenza selezionabile verrà attraversata, verificando per ogni cella se è intersecata dal punto di click del mouse. Se una cella intersecante selezionata è riscontrata,

diverrà selezionata, a meno che il metodo **isToggleSelectionEvent** ritorna true, nel cui caso la cella selezionata più in alto, intersecata, è rimossa dalla selezione (di solito con uno Shift-Click.)

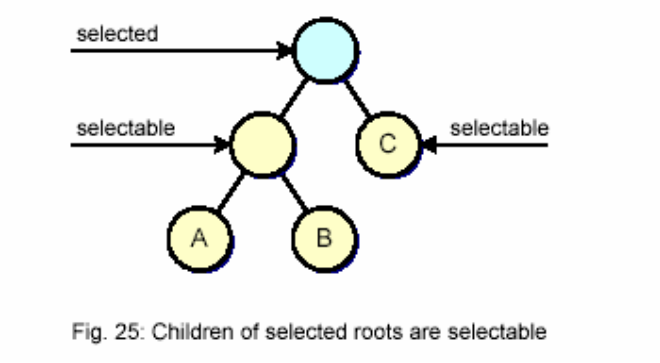

Fig. 25: Children of selected roots are selectable

Quando una cella è aggiunta alla selezione, i suoi figli sono aggiunti tra le celle selezionate e la prossima cella nella sequenza (la cella selezionata non è rimossa dalla sequenza). Perciò, il sistema prima seleziona i figli della cella più in alto, e dopo continua a selezionare le celle sottostanti e i loro rispettivi figli, su selezione dei loro padri.

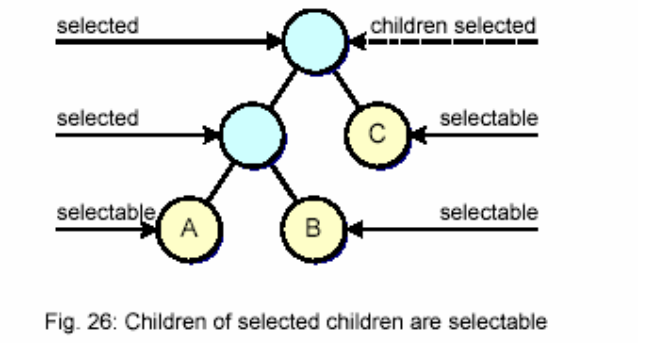

Fig. 26: Children of selected children are selectable

Quando una cella è rimossa dalla selezione, tutti i suoi figli sono rimossi dalla sequenza. La cella stessa è deselezionata, ma ancora contenuta nella lista dei selezionabili. Perciò una cella lascia la lista di selezionabili solo se suo padre è deselezionato, ciò accade per le celle non-root, poiché le roots non hanno padri.

Poiché ogni vista può fornire un layering differente alle celle, e la modalità di selezione non è a conoscenza del layering della vista, la lista di selezione che viene ritornata dal modello di selezione è più precisamente definita come un set di candidati alla selezione. Il set è poi convertito in una sequenza di celle ordinate in base al lyering della vista rispettando le norme sopra stabilite.

4 La Vista

La vista in Jgraph è differente dalle altre classi Swing che esibiscono la parola *view* nelle loro signature. La differenza è che la vista in Jgraph è con stato, cioè contiene informazioni che sono unicamente contenute nella vista. L'oggetto **GraphView** e le istanze di **CellView** costituiscono il modello della vista di un grafo, che mantiene una rappresentazione interna del modello del grafo.

4.1 La Vista del Grafo

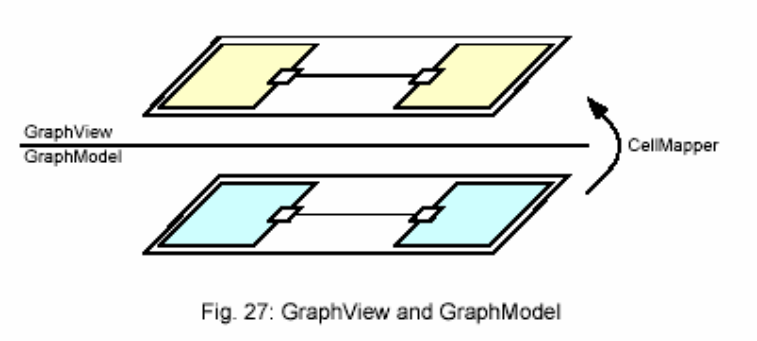

Fig. 27: GraphView and GraphModel

L'oggetto **GraphView** mantiene le viste delle celle, una per ogni cella del modello. Queste viste sono tenute in due strutture, un array di port view, e una lista che contiene le viste delle celle root. Così le porte e le altre celle sono conservate in due strutture di dati differenti nel graph view, sebbene nel modello esse sono tenute in un'unica struttura. La graph view inoltre ha un riferimento ad una tabella hash, che serve a fornire un mappaggio dalla cella alla sua vista. Il mappaggio inverso non occorre in quanto le cell views puntano alla loro cella corrispondente.

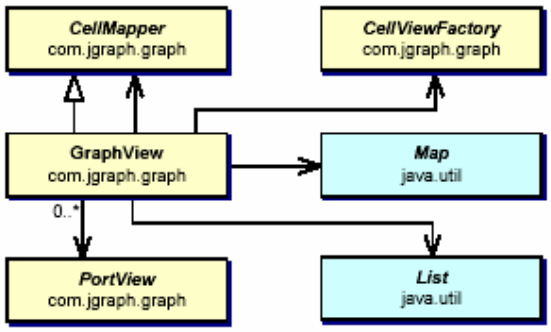

Fig. 28: GraphView class and static relations

GraphView estende **CellMapper**, ha un riferimento a un **CellMapper** (di solito **this**), e ad una **CellViewFactory,** in più istanzia un array di **PortView,** una **List** che mantiene le viste delle root, e una map che si usa per implementare l'interfaccia **CellMapper**.

I figli possono essere accessi attraverso l'interfaccia **CellView**, e non sono accessibili dall'oggetto **GraphView**. La creazione di nuove viste è differita a **CellViewFactory.** Per default, Jgrah fornisce questa factory, ossia è l'oggetto Jgraph l'ultimo responsabile di creare le viste.

La rappresentazione della graph view di un grafo è differente dalla struttura interna del modello, poiché e disegnata con l'operazione di visualizzazione in mente.Per esempio la vista, al contrario del modello, fa differenza tra figli che implementano l'interfaccia **Port** e altri figli. I figli port sono trattati in maniera separata, sia come figli sia al fine della memorizzazione e sono tenuti in un array separato.

Questo poiché l'insieme di porte disponibili è spesso acceduto indipendentemente da tutto il resto, per esempio quando il grafo è disegnato con con la port visibility settata a true, o quando la sorgente e la destinazione di uno spigolo è modificata interattivamente, in tal caso le nuove porte devono essere trovate o illuminate. Per trovare la port in una specifica locazione, la graph view cicla tra l'array delle porte in una vista e verifica se una porta interseca il punto specificato, se è così, restituisce questo punto.

Anche se le porte sono sempre visibili e disegnate sul top delle celle, l'ordine in cui sono immagazinate riflette il layering delle corrispondenti celle nella vista. Ciò significa, se due porte sono sul top una dell'altra, la porta che appartiene alla cella più in alto sarà selezionata per prima.

4.2 Il Mappatore di Celle (Cell Mapper)

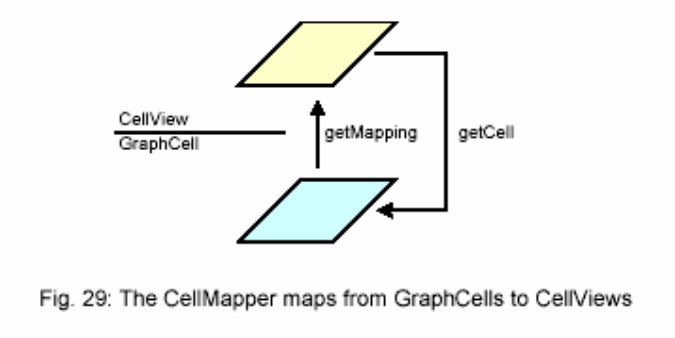

Fig. 29: The CellMapper maps from GraphCells to CellViews

La classe **GraphView** implementa l'interfaccia **CellMapper**, che definisce il mappaggio da celle a viste. L'interfaccia fornisce due metodi, uno restituisce le vista di cella per una data cella e opzionalmente crea la vista se non esiste, usando la **CellViewFactory** e una che associa una data cella con una vista di cella appena creata.

La graph view fornisce in più altri utili metodi, non altrimenti richiesti da questa interfaccia, un metodo per mappare un array di celle ad un array di viste, uno per ottenere tutti i discendenti di una specifica vista (senza includere le porte), e uno per ottenere i confini visuali da un array di viste.

L'interfaccia **CellMapper** è usata in Jgraph per incapsulare la parte più importante di un oggetto **GraphView**, ossia il mappaggio da **GraphCell** a **GraphViews**. (In più permette al mappaggio della graph view di essere scambiato a run-time, che è necessario per la live-preview, di cui dopo). I mappaggio inverso non è necessario in quanto ciascuna istanza **CellView** ha un riferimanto alla corrispondente istanza di **GraphCell**.

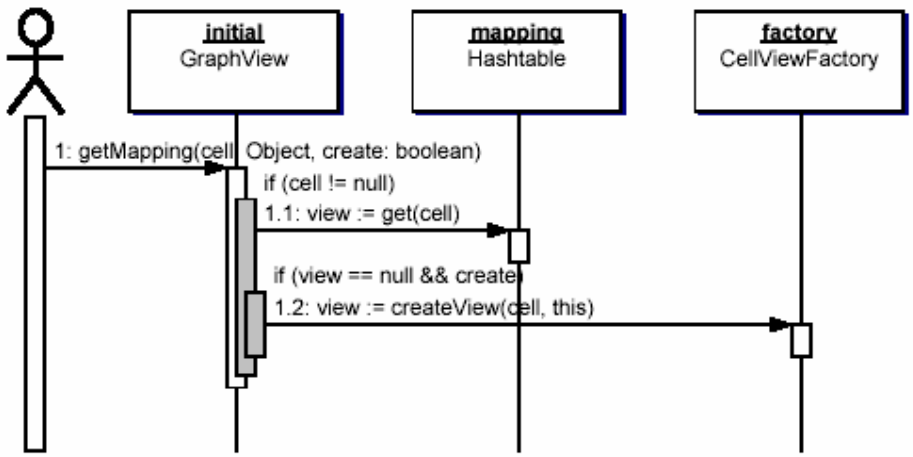

Fig. 30: CellViews may be created automatically if they do not exist

Quando una cella è osservata nella graph view per trovare la sua corrispondente cell view, un accesso ad una tabella hash è eseguito, usando l'hash code della cella, che è fornito come per ogni **Object**, come key. Se la cella non è associata, ossia se la hash table ritorna null, la vista è creata basandosi sull'argomento **create**, usando la **CellViewFactory**.

4.3 CellView Factory

L'associazione tra una cella e la sua vista è stabilita durante la creazione della vista, attraverso l'uso di **CellViewFactory.** Questo perché la creazione di una vista di cella comporta anche la creazione delle viste dipendenti, come quelle dei figli, o della sorgente e destinazione di uno spigolo. Perciò l'associazione tra la cella iniziale e la sua vista deve essere disponibile quando le viste dei figli sono create (poiché le viste delle celle figlie mantengono un riferimento alla vista della cella padre, le viste degli spigoli hanno un riferimento alle viste delle porte sorgente e destinazione, che sono ottenute usando l'oggetto **CellMapper,** che viene passato al metodo **createView** durante la creazione di una vista.)

4.4 Cell Views

Le viste di cella sono usate per memorizzare il pattern geometrico del grafo, e anche per associare il renderer, un editor e un handle di cella con la creazione della vista di cella. Il renderer è responsabile del disegno della cella, l'editor è usato per l'editing in-place, e l'handle di cella è usato per editing più sofisticati. Per efficienza, il pattern *Flyweight* è usato per condividere il renderer tra più istanze di una specifica classe **CellView**. Questo si ottiene rendendo **static** il riferimento al renderer. L'editor e l'handle di cella, invece, sono creati al volo.

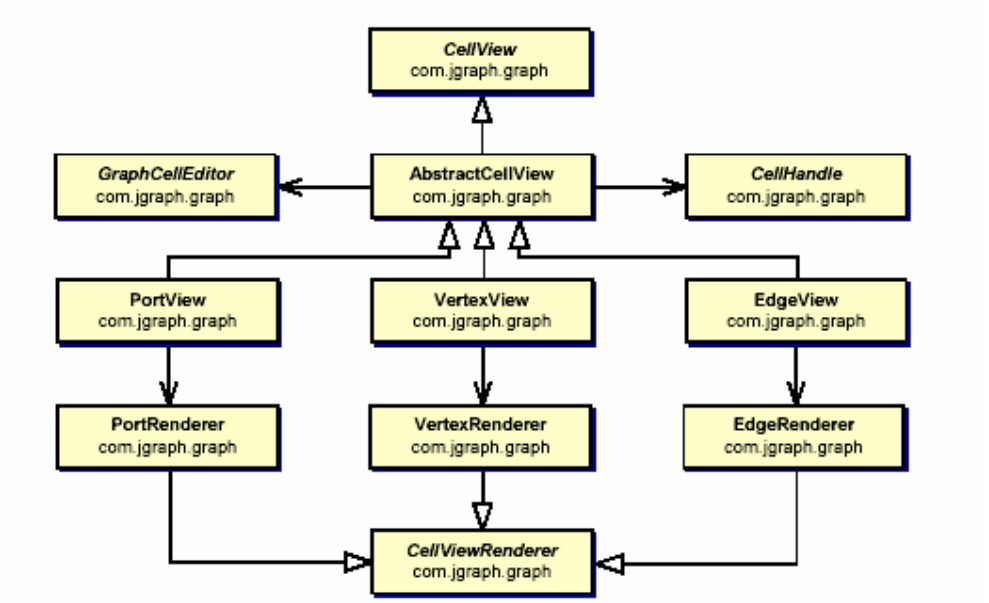

Fig. 31: CellView interface, default implementations and static relations

VertexView, EdgeView e **PortView** estendono **AbstractCellView**, che di per se estende **CellView**. **AbstactCellView** ha un riferimento statico a **GraphCellEditor** e a **CellHandle**. Ogni vista concreta ha un riferimento statico alla sua corrispondente sottoclasse di **CellViewRenderer**, che è usata per disegnare la cella.

4.4.1 L'interfaccia CellView

L'interfaccia **CellView** è usata per accedere le strutture di grafo e di gruppo del grafo e per conservarne gli attributi. **CellView** fornisce accesso alle viste padre e figlie e a un **CellViewRenderer**, ad un **GraphCellEditor** e ad un **CellHandle**. Il metodo **getBounds** restituisce i confini della vista, e il metodo **intersects** si usa per la hit-detection.
Il metodo **refresh** viene messaggiato quando la cella corrispondente è cambiata nel modello, così pure il metodo **update**, ma quest'ultimo è invocato anche quando una cella dipendente varia o quando la vista ha bisogno di essere aggiornata (per esempio durante la live-preview).

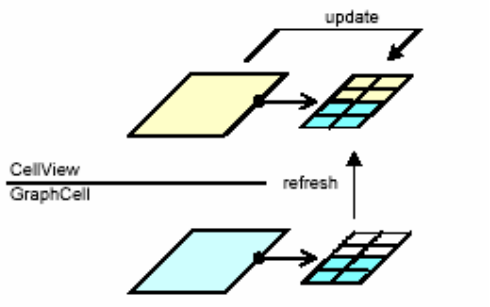

Fig. 32: CellView update and refresh

Il metodo **update** è un buon posto per implementare attributi automatici, come il routing degli spigoli, ossia attributi che sono calcolati in base ad altri attributi o in base alla geometria del grafo. Gli attributi di una vista di cella sono acceduti usando il metodo **getAttributes**, con eccezione dei bounds, che sono cachati e acceduti usando **getBounds**.

4.4.2 Le Implementazioni di Default di CellView

Per default, l'oggetto Jgraph, che agisce come **CellViewFactory** è in grado di ritornare cell views per ognuna delle implementazioni dell'interfaccia **GraphCell**. La vista di cella è creata in base al tipo di cella passata. Per oggetti che implementano **Edge**, sono create **EdgeView**, e per oggetti che implementano **Port** sono create **PortView**. Per gli altri oggetti sono create **VertexView**.

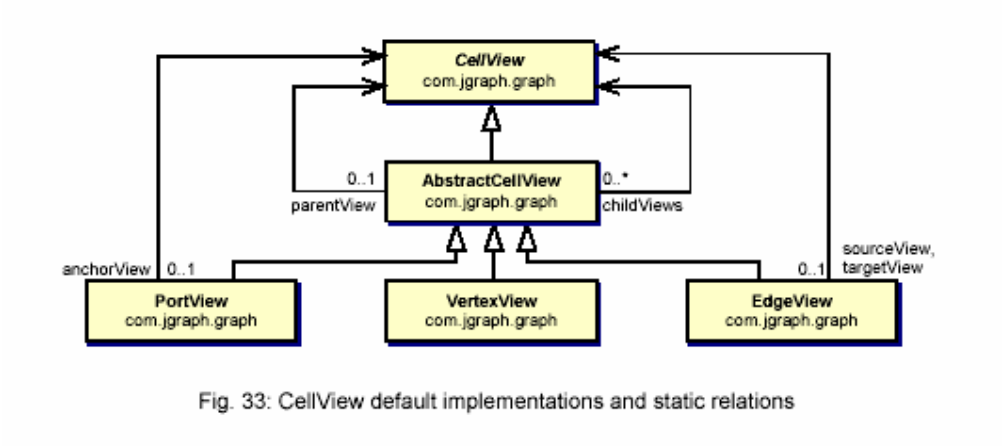

Fig. 33: CellView default implementations and static relations

AbstractCellView ha un riferimento alla **CellView** padre e ha un array di **CellView** che rappresentano le viste dei figli. **EdgeView** ha un riferimento alle **CellView** delle porte sorgente e destinazione, mentre **PortView** le mantiene all'anchor della **PortView** stessa. In contrapposizione al set di spigoli di una **DefaultPort**, la **PortView** non fornisce accesso al set di spigoli attaccati.

4.4.2.1 AbstractCellView

La classe **AbstractCellView** fornisce le funzionalità di base alle sue sottoclassi. Per esempio implementa il metodo **getRendererComponent** per creare un renderer, configurarlo e restituirlo. Il renderer concreto è ottenuto usando una chiamata ad un metodo interno astratto **getRenderer**, che deve essere implementato dalle sottoclassi per ritornare un renderer specifico.

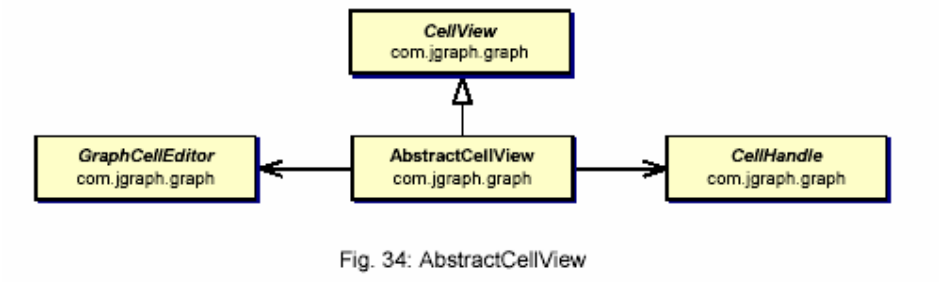

Fig. 34: AbstractCellView

Il metodo **intersects**, che è usato per implementare la hit-detection, usa il metodo **getBounds** per verificare se un dato punto interseca una vista di cella. La classe **EdgeView** forza questa implementazione di defualt, poiché nel contesto di uno spigolo, un hit dovrebbe solo essere ottenuto se il mouse è sopra la sagoma dello spigolo (o la sua etichetta), ma non se l'hit avviene dentro i confini e fuori dalla sagoma.

4.5 Cambiare la Vista

Fig. 35: Nested map for the graph view's edit method

Usando il metodo **edit**, gli attributi dalle cell views possono essere cambiati. L'argomento è una mappa dalle views agli attributi. I metodi **toBack** e **toFront** sono usati per cambiare il layering delle viste di cella.

4.6 Il Graph Context

La classe **GraphContext** prende nome dal fatto che ciascuna selezione in un grafo può possibilmente avere un certo numero di celle dipendenti, che non sono selezionate. Queste celle, devono essere trattate in un modo speciale per ottenere una corretta live-preview, ossia una vista esatta di come il grafo si presentera dopo che la corrente transazione abbia avuto effetto. Ciò perché per esempio uno spigolo può cambiare posizione quando le sue porte sorgente e destinazione sono mosse, anche se lo spigolo stesso non è selezionato.

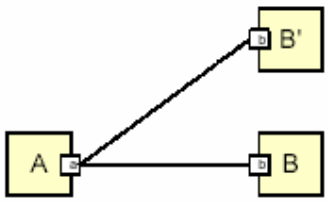

Fig. 36: GraphContext for the moving of vertex B

Per esempio, quando si muove il vertice B nel grafo di sopra, la sua porta e lo spigolo attaccato devono essere duplicati e ridisegnati per riflettere la nuova posizione del vertice, anche se lo spigolo e la porta stesse non sono selezionate. Perciò, la selezione consiste del solo vertice B, la porta e lo spigolo sono duplicati poiché parti del contesto di questa selezione. (Nel caso in cui l'utente tiene premuto il tasto control mentre si muove una cella, il che viene interpretato come clonaggio, le celle non selezionate non sono visibili, poiché esse non saranno clonate quando il pulsante del mouse è rilasciato.)

Il termine contesto è usato per descrivere il set di celle che sono indirettamente modificate dalla modifica, ma che non sono esse stesse parte della selezione. L'oggetto **GraphContext** è usato per calcolare e gestire questo set. Di più, l'oggetto può essere usato per disconnettere un set di spigoli dal suo grafo originale, ossia da tutte le porte che non fanno parte del **GraphContext**. (Essere parte del graph context significa sia che una cella o uno dei suoi ancestori è selezionato sia che la cella è parte del contesto di questa selezione.)

4.6.1 La Costruzione

Per creare un oggetto **GraphContext**, la selezione corrente è passata al costruttore. Nel costruttore, il set di discendenti delle celle selezionate è calcolato usando il metodo del modello **getDescendants**, e memorizzato all'interno dell'oggetto **GraphContext**. Dopo, il set di spigoli che è attaccato a questi discendenti è calcolato usando il metodo del modello **getEdges** e il set di discendenti è sottratto da questo, risultando nel context, ossia il set di celle che è attaccato, ma ne selezionato ne avente ancestori selezionati.
Avendo questi due set per le mani, il graph context può creare le viste temporanee per le celle selezionate a per il loro contesto. Le viste temporanee sono tipicamente usate per fornire una live-preview per le modifiche che sono correntemente in-progress, come un operazione di move o resize. Per tali modifiche, le viste temporanee sono create usando il suddetto meccanismo, e poi cambiato in-place, finchè l'utente rilascia il pulsante del mouse, in tal caso i cambi sono applicati agli attributi originali usando il metodo **applyMap**. Se l'utente preme esc durante il cambio, allora il cambio è abortito e perciò gli attributi iniziali non vengono modificati in questo caso.

4.6.2 Le Viste Temporanee

La viste temporanee sono create in due passi : primo, le vista temporanee delle celle selezionate sono create e associate con le corrispettive celle usando la hash-table locale. Poi, le viste temporenee del context sono create, usando l'istanza di **GraphContext** come **CellMapper** e sono anche conservate nella stessa hash-table locale.

Il mappaggio nel contesto grafico usa la hash-table locale per ottenere i riferimenti alle viste temporanee. Ossia, le viste create dal graph context referenziano le viste temporanee come loro padri, figli o sorgenti e destinazioni, se tali viste temporanee esistono. (Le viste temporanee sono create in modo lazy, quando è necessario, in base alla cella contenuta o nel set di discendenti oppure nel contesto, e memorizzate nella hash-table locale per la creazione.)

Perciò, in contrasto alla graph view, questo mappaggio di cella crea unicamente una nuova (temporanea) vista di cella se la cella è parte del contesto, e perciò, le risultanti viste referenziano le viste di cella originarie se le corrispondenti celle non sono parte del context.
Queste viste temporanee sono usate per la live-preview, che è spiegata nel prossimo capitolo, poiché è collegata con **CellHandles**, che è usato nel UI-delegate (il "controllore").

5 Il Controllore

Il control definisce il mappaggio da eventi utente ai metodi dei modelli di grafo, di selezione e di vista. E' implementato in maniera dipendente dalla piattaforma nel UI-delegate e di solito durante l'editing in-place, il cell handling e l'aggiornamento dello schermo. Essendo l'unico oggetto che è scambiato quando cambia il L&F, il delegato UI specifica anche la parte di L&F specifica per il rendering.

5.1 UI-Delegate

La classe astratta **GraphUI** e le sue implementazioni di default, **BasicGraphUI**, costituiscono il delegato UI di Jgraph. Aldilà dei listener di evento che tracciano l'input dell'utente, il delegato UI è anche usato per disegnare il grafo e aggiornare lo schermo quando la selezione cambia. Quando un grafo viene disegnato, il delegato UI cicla tra le viste e le disegna attraverso i loro rispettivi renderer.

5.1.1 L'Interfaccia GraphUI

La classe astratta **GraphUI** definisce i metodi per un oggetto che può essere usato come un delegato UI per Jgraph, mentre **BasicGraphUI** fornisce le implementazioni di default per tali metodi. **BasicGraphUI** può anche essere sottoclassato, per implementare l'UI per uno specifico L&F, ma l'implementazione di default già contiene alcune colorazioni tipiche di alcuni L&F.

5.1.2 L'Implementazione di Default di GraphUI

La classe **BasicGraphUI** mantiene un riferimento ad un **RootHandle**, che è usato per reagire agli eventi del mouse. Il listener del mouse, che chiama l'handler, è una classe annidata del delegato UI. Definisce il conportamento di base di Jgraph per quanto riguarda gli eventi del mouse e implementa le due funzionalità principali : selezione ed editing. La selezione include la selezione singola di cella e la selezione marquee. L'editing può essere o basato sull'interfaccia **CellEditor** per l'in-place editing o l'interfaccia **CellHandle** per l'handling della cella.

BasicGraphUI crea gli handle delle root, mentre gli handle delle altre celle sono creati dalle view. L'handle di root è responsabile di muovere la selezione, i suoi figli sono usati per cambiare la sagoma delle celle. Un set di collegamenti di default alla tastiera è supportato dal delegato UI, c'è una lista in tabella (il numero di click che attivano l'in-place editing può essere cambiato usando il metodo **setEditClickCount** di Jgraph).

Double-click / F2	Starts editing current cell
Shift-Click	Forces marquee selection
Shift-Select	Extends selection
Control-Select	Toggles selection
Control-Drag	Clones selection
Shift-Drag	Constrained drag

Table 1: JGraph's default keyboard bindings

5.2 Renderers

I renderers non disegnano le celle ; loro disegnano l'oggetto che implementa l'interfaccia **CellView.** Per ogni cella del modello, esisterà esattamente una vista di cella in ciascuna vista di grafo, che ha la sua rappresentazione interna del modello del grafo. I renderer sono instanziati e referenziati dalle viste di cella.

I renderer sono basati sull'idea della classe **TreeCellRendere** [16] di swing, e sul pattern *Flyweight* (vedi [17]). L'idea di base dietro questo pattern è "l'uso condiviso di un largo numero di oggetti a grana fine per l'efficienza".

Avere componenti separati per ciascuna istanza di **CellView** sarebbe proibitivo ed costoso, il componente è condiviso tra tutte le viste di cella di una certa classe. Una vista di cella perciò conserva solo lo stato del suo componente (come il colore, la grandezza, ecc..), mentre il renderer ha il componente e il codice di disegno (per esempio una istanza di **Jlabel**.)

La cell views sono disegnate configurando il renderer e disegnandolo nel **CellRendererPane** [19], che può essere usato per disegnare i componenti senza la necessità di aggiungerli come nel caso dei contenitori. Configurare un renderer significa ottenere lo stato della vista di cella e applicarlo al renderer. Per esempio nel caso di un vertice, che usa una istanza di Jlabel come renderer, l'etichetta della cella si ottiene con **graph.convertValueToString** e settata usando **setText** sul renderer, insieme con altri attributi che fanno lo stato del vertice.

I renderer in Jgraph sono usati in analogia ai renderer di Jtree, solo che Jgraph fornisce più di un renderer, ossia uno per ogni tipo di cella. Così, Jgraph fornisce un diverso renderer per i vertici, gli spigoli e le porte. Per ogni sottotipo dell'interfaccia **CellView**, forzando il metodo **getRenderer** vi si può associare un nuovo renderer. Il renderer dovrebbe essere static per permettergli di essere condiviso tra instanze multiple di una classe.

Il rendere stesso è tipicamente un istanza di classe **Jcomponent**, con il metodo **paint** forzato che disegna la cella, in base agli attributi della vista di cella che viene ad esso passata. Il renderer implementa anche l'interfaccia **CellViewRenderer** che fornisce il metodo **getRendererComponent** per configurare il renderer.

5.2.1 L'Interfaccia CellViewRenderer

Questa interfaccia fornisce due metodi : uno per configurare e restituire il renderer, l'altro per verificare se un dato attributo è supportato. Gli argomenti del primo metodo sono usatii per specificare la vista della cella e il suo stato, ossia se è selezionato, se ha il focus e se deve essere disegnato come una live-prview o deve essere illuminato.
L'argomento di live-preview è true se il renderer è ussato nel contesto di live-preview, che richiede un ridisegno più veloce. In tal caso, il renderer può decidere di non disegnare la cella in qualità piena in base al costo di tale disegno. L'argomento passato a questo metodo può non essere dedotto dalla vista di cella ; essi sono determinati dal delegato UI, dal modello di selezione o dal context nel quale il renderer è usato.

5.2.2 Le Implementazioni di default di CellViewRenderer

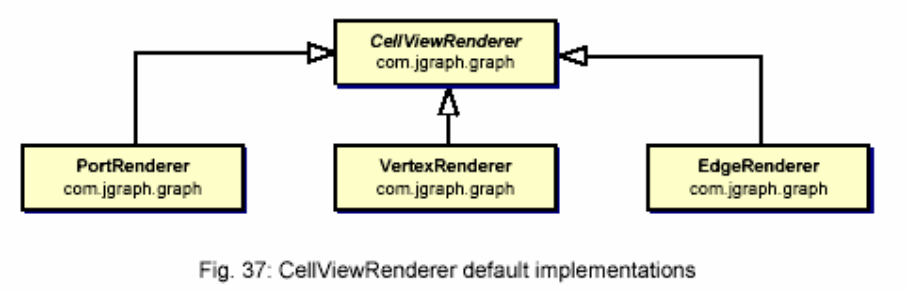

Fig. 37: CellViewRenderer default implementations

Le tre implementazioni di default della interfaccia **CellViewRenderer** sono le classi **VertexRenderer, EdgeRenderer** e **PortRenderer. Vertex Renderer** è una estensione di Jlabel che offre il codice per disegnare l'etichetta, l'icona,il bordo e il background del vertice. Il metodo **paint** è forzato solo per disegnare il bordo di selezione.

Gli altri renderer sono sottoclassi di **Jcomponent** poiché fanno dei rendering speciali, che non sono direttamente offerti dalle estensioni di **Jcomponent**. **EdgeRenderer** usa la classe **Graphics2D** e l'interfaccia **Shape** per disegnare le etichette, gli spigoli e le decorazioni. **PortRenderer** fornisce il codice per disegnare una piccola croce o una porta illuminata nel caso in cui l'argomento di highlight è su true.

Il flag **preview** è ignorato nel contesto del **PortRenderer** poiché nelle live-preview, le porte non sono mai disegnate e perciò il flag non è mai true.

5.3 Editor

L'in-place editing è ottenuto attraverso un set di metodi che sono forniti dal delegato UI. Internamente, l'oggetto che è usato come un editor è ritornato dalla cell view, proprio come fa il renderer. Diversamente dal renderer, c'è solo un **GraphCellEditor** per tutte le celle, anche se il design permette a differenti editor di essere specificati per ciascun tipo di vista di cella, anche per ciascune istanza di cell view.

Come per l'interfaccia **CellViewRenderer,** l'interfaccia**GraphCellEditor** fornisce un metodo che è usato per ritornare un editor configurato per la specifica cella. L'argomento a questo metodo non è **CellView,** come nel caso di **CellViewRenderer** ; è una cella invece, che è un **Object**. Questo sottolinea il fatto che l'etichetta è tipicamente memorizzata nel modello non nella vista.

L'editor è messo dentro il grafo dal delegato UI, che è anche responsabile di rimuoverlo da lì, e di notificare i cambi avvenuti al modello. Perciò, per un edititng customizzato, può essere necessario fornire un delegato UI customizzato, forzando i metodi protetti **startEditing** e **completeEditing** della classe **BasicGraphUI**.

5.4 Cell Handles

I cell handles sono un nuovo concetto che non è usato altrove in swing. Ciò perché in swing, il solo modo per editare le celle è con l'in-place editing. Non è possibile cambiare i confini (bounds) della cella o muovere la cella in una posizione arbitraria. Sono invece il delegato UI e lo stato della cella a determinare la locazione della cella. In Jgraph invece, la posizione e la grandezza sono memorizzati tra gli attributi e non dipendono dallo stato del grafo. Perciò, un modo deve essere fornito per cambiare gli attributi in un modalità orientata alla transazione.

Gli handles sono basati sul pattern *Composite*, dove una root fornisce accesso ai figli in base ad una interfaccia comune : **CellHandle**. Il delegato UI crea una **RootHandle** in un cambio di selezione. Il root handle usa il metodo **getHandle** di **CellView** per creare gli handle dei figli.

Mentre l'handle di root è attivo, l'handle del mouse interno al delegato UI messaggia i suoi metodi e il root handle delega il controllo al corretto sotto handle o assorbe gli eventi per muovere la selezione.

5.4.1 Live-Preview

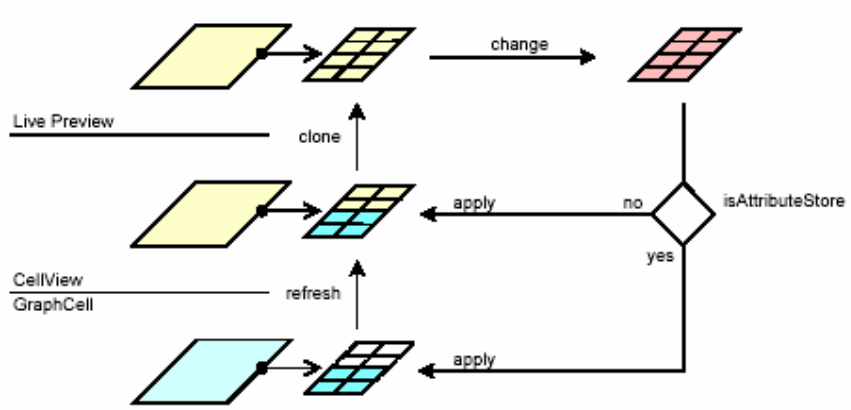

Fig. 38: Live-preview

Durante un cambio al grafo, i nuovi attributi sono mostrati usando le viste temporanee delle celle, che sono state precedentemente create usando l'oggetto **GraphContext**. Questo è chiamato live-preview, indicando che il grafo è sovrapposto con il cambio, così che l'utente possa vedere come il grafo apparirà quando il mouse sarà rilasciato. (Il cambio può essere cancellato premendo esc).

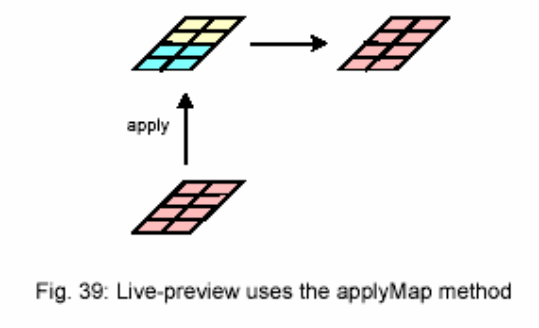

Fig. 39: Live-preview uses the applyMap method

Se il modello è un attribute store, allora il cambio è eseguito sul modello. Altrimenti, il cambio è eseguito sulla vista. In ambo i casi, il metodo **applyMap** è usato, fornito dalla classe **GraphConstants**.

5.4.2 L'Interfaccia CellHandle

L'interfaccia **CellHandle** è molto simile ad un **MouseAdaptor**, fornendo tutti i metodi che sono usati per gestire gli eventi del mouse. In più, l'interfaccia definisce due metodi per disegnare l'handle, ossia il metodo **paint,** che è usata per disegnare la parte statica (l'handle attuale) e il metodo **overlay** che è usato per disegnare la parte dinamica (la live-preview), usando un metodo di disegno veloce a XOR.

5.4.3 Le implementazioni di Default di CellHandle

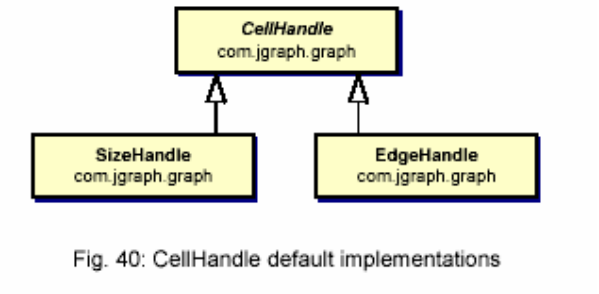

Fig. 40: CellHandle default implementations

Le implementazioni di default di **CellHandle** sono le classi **SizeHandle, EdgeHandle e RootHandle**. L'handle di root è responsabile di muovere le celle, l'handle di size di modificarne le dimensioni, e l'handle di spigolo permette di connettere e disconnettere spigolo e di aggiungere, modificare e rimuovere punti individuali sullo spigolo.

5.5 GraphTrasferable

Il delegato UI fornisce una implementazione della classe **TransferHandler** che è responsabile di creare il **Transferable** in base alla selezione corrente.

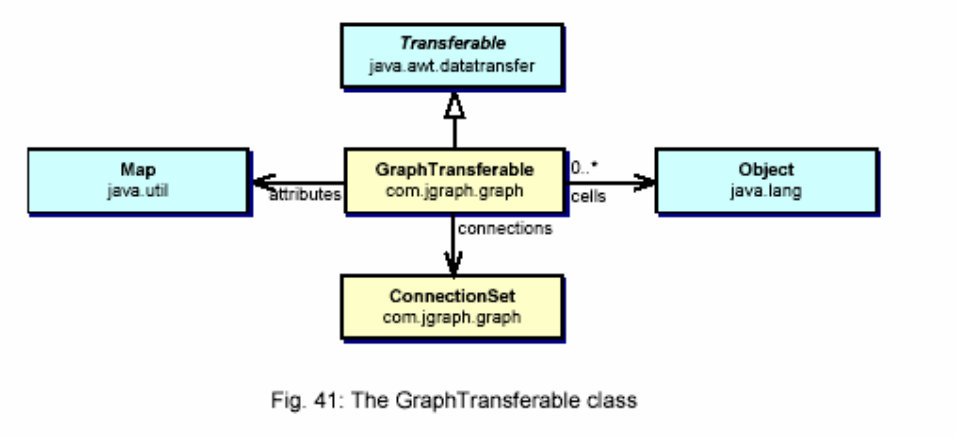

Fig. 41: The GraphTransferable class

Il transferable è rappresentato dalla classe **GraphTransferable,** che ha un riferimento ad un **ConnectionSet,** alle celle e ai loro corrispondenti attributi di vista. Il connection set è creato usando il *metodo factory static* che è stato sottolineato prima. I dati possono essere ritornati come oggetti serializzati, o come text o come HTML. Negli ultimi due casi l'etichetta della cella selezionata è restituita o un stringa vuota, se il transferable contiene più di una cella.

5.6 La Selezione Marquee

La selezione marquee è la capacità di selezionare una regione rettangolare usando il mouse, abilità non fornita da swing. La classe **BasicMarqueeHandler** è usata per implementare questo tipo di selezione. Da un punto di vista architetturale, l'handler marquee è analogo all'handler di transfer, poiché è un listener di alto livello che è invocato dai listener di basso livello, come quelli del mouse, che sono installati nel delegatoUI.
Per ciò che concerne i suoi metodi, l'handle di marquee è più simile all'handle di cella, poiché l'handle di marquee tratta gli eventi del mouse e permette modi di disegno addizionali e l'overlyering del marquee. (Il marquee è un rettangolo che costituisce la regione da selezionare.)

5.7 Il Modello degli Eventi

In Jgraph, il modello del grafo, il modello di selezione e la vista del grafo possono dispacciare eventi. Gli eventi dispacciati dal modello possono essere suddivisi in :

- Notifiche di cambio
- Supporto per la cronologia dei cambi

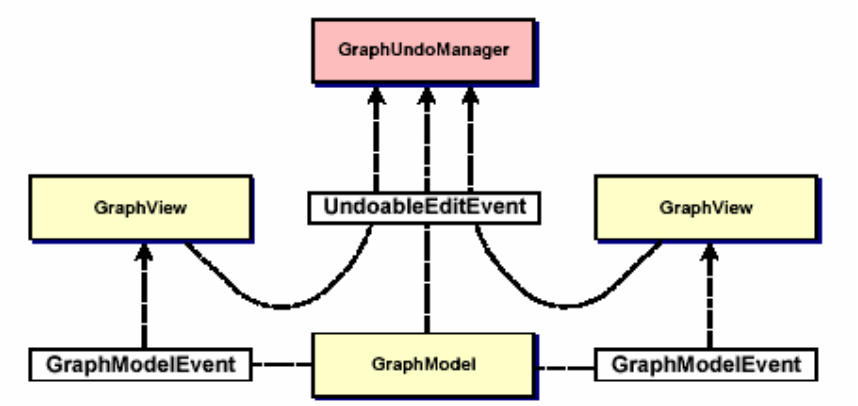

Fig. 42: JGraph's event model

L'interfaccia **GraphModelListener** è usata per aggiornare la vista e ridisegnarla, mentre l'interfaccia **UndoableEditListener** per aggiornare la cronologia. Questi listener possono essere registrati o rimossi usando il rispettivo metodo del modello.

Il modello di selezione dispaccia **GraphSelectionEvent** al **GraphSelectionListener** che è stato registrato, per esempio per aggiornare lo schermo con la selezione corrente. Il modello degli eventi di vista è basato sulle classi **Observer/Observable** per ridisegnare il grafo e fornire anche il supporto per l'undo, che usa l'implementazione del modello del grafo.

Un meccanismo addizionale di notifica tra modello e vista, abilita un singolo modello ad avere più di una vista associata. Nota che la vista non implementa da se l'interfaccia **GraphModelListener**, ma viene messaggiata dal listener del modello che è parte del delegato UI, passando informazioni sul cambio come un argomento.

Ogni cambio è eseguito o sul modello o su una vista specifica o è una combinazione di entrambi. Non esistono cambi inerenti a più viste contemporaneamente, eccetto implicitamente quando c'è un cambio al modello. Ciò porta ad una notificha separata tra la vista locale e il suo delegato UI da ridisegnare. La classe **GraphView** implementa l'interfaccia **Observable** e il **BasicGraphUI** fornisce una istanza di **Observer** che può essere registrata con la graph view e ha il compito di ridisegnare il grafo in caso di modifica alla sola vista.

5.7.1 Notifica di Cambio

Durante inserzioni, rimozioni e modifiche alle celle nel modello, il modello costruisce un oggetto che esegue e descrive il cambio. Poiché l'esecuzione e la descrizione di tale cambio sono strettamente correlati, essi sono conservati nello stesso oggetto. (Durante un cambio i *metodi factory* delle classi **ParentMap** e **ConnectionSet** sono usati per costruire il rispettivo oggetto che descrive il cambio).

L'oggetto è in seguito inviato ai listener del modello di modo che questi possano ridisegnare le regioni affette del grafo, e rimuovere, aggiungere o aggiornare le viste di celle delle celle cambiate.I listener possono anche volersi assicurare di altre proprietà, come per l'auto-sizing, in tal caso la dimensione preferita è calcolata quando la cella è modificata.

Diversamente dalla interfaccia **TreeModelListener**, che fornisce tre metodi che vengono messaggiati, uno per l'inserzione, uno per la rimozione, uno se le celle sono cambiate, l'interfaccia **GraphModelListener** fornisce solo un metodo che è informato di ogni cambio.
(L'ultima versione di **TreeModelListener** ha in più un metodo chiamato **treeStructureChange**).

C'è una sottile conseguenza quando avvengono cambi compositi, che possono inserire, rimuovere e cambiare celle allo stesso tempo. Per i listener di Jtree, questi cambi sono gestiti da parti differenti del codice di gestione, risultando in fino a quattro separati aggiornamenti per un cambio composito, mentre in Jgraph, l'intero cambio composito è gestito in un solo metodo, ottenendo un singolo aggiornamento che è molto più efficiente, anche se tali cambiamenti compositi sono rari.

I listener del modello sono notificati usando oggetti che implementano l'interfaccia **GraphModelEvent,** che restituisce un oggetto che implementa l'interfaccia **GraphChange**. Quest'ultima interfaccia è usata per descrivere un cambio, ossia essa restituisce le celle che sono state aggiunte, rimosse o cambiate. Una classe annidata del modello, che contiene anche il codice per eseguire e fare l'undo di un cambio, implementa l'interfaccia **GraphChange.**

5.7.2 Il Supporto per l'Undo

Il supporto per l'undo, ossia l'immagazinamento dei cambi che sono stati eseguiti precedentemente, è di solito implementato al livello applicativo. Ciò vuol dire che Jgraph stesso non fornisce un cronologia, ma fornisce solo le classi e i metodi per supportarla a livello applicativo. Questo accade perché la cronologia richiede molto spazio di memoria e dipendente da quanti passi sono conservati (il che è un parametro dell'applicazione). In più la cronologia non può essere sempre implementata e non è sempre desiderabile.

L'oggetto **GraphChange** è inviato al **UndoableEditListener** che è stato registrato con il modello. L'oggetto perciò implementa le interfacce **GraphChange** e **UndoableEdit** . L'ultima di queste è usata per implementare il supporto all'undo e fornisce i metodi **undo, redo.** (Il codice per eseguire un undo o redo del cambio è conservato dentro l'oggetto e viaggia verso i listener.)

5.7.3 Undo-Support Relay

Insieme al modello, la graph view usa anche il codice che il modello fornisce per notificare il suo undo listener di un cambio che può accettare l'undo. Ciò può essere fatto perché ogni view tipicamente ha un riferimento al modello, mentre il modello non ha riferimenti alle sue viste.
(L'interfaccia **GraphModel** allows relaying **Undoable Edits** usando il quarto argomento al metodo **edit**).

La classe **GraphView** usa il supporto all'undo del modello per passare **UndoableEdits** che crea per il **UndoableEditListener** che si è registrato col modello. Ancora, l'oggetto che viaggia verso i listeners contiene il codice per eseguire il cambio sulla view e anche quello per fare undo o redo su tale cambio.
Questo da il vantaggio che **GraphUndoManager** deve unicamente essere attaccato al modello, invece che al modello e ad ogni view.

5.7.4 GraphUndoManager

Geometrie separate, che sono conservate indipendentemente dal modello, portano a cambi che sono possibilmente visibili solo in una vista, e non influenzano le altre view o lo stesso modello.
Le altre viste sono inconsapevoli del cambio e se una di loro chiama undo, ciò deve essere tenuto in conto.
Una estensione del **UndoManager** di Swing nella forma di **GraphUndoManager** è disponibile per undo e redo dei cambi nel contesto di viste multiple.
GraphUndoManager forza i metodi di **UndoManager** con un argomento in più, che permette di specificare la vista chiamante come context per le operazioni di undo/redo.
Questo contesto è usato per determinare l'ultima o la prossima transazione rilevante nel rispetto della vista chiamante. Rilevante qui significa visibile, ossia tutte le transazioni che non sono visibili nella vista chiamante fanno undo o redo implicitamente, finchè la prossima o l'ultima transazione è trovata per il contesto.

Come esempio consideriamo la seguente situazione : Due viste condividono lo stesso modello, che non è un attribute store. Ciò vuol dire che, ciascuna vista può cambiare indipendentemente, e se il modello cambia, entrambe le viste sono aggiornate correttamente. Il modello notifica le sue viste se le celle sono aggiunte o rimosse o se la connessione del grafo è modificata, implicando che o la sorgente o la destinazione di uno o più spigoli è cambiata. Tutti gli altri casi sono transazioni sulla sola vista, come per esempio se le celle sono mosse, ridimensionate o se un punto viene aggiunto, modificato o rimosso dallo spigolo. Tutte le viste eccetto la vista sorgente sono inconsapevoli di tali transazioni only-view , poiché tali transazioni sono visibili solo nella vista sorgente.

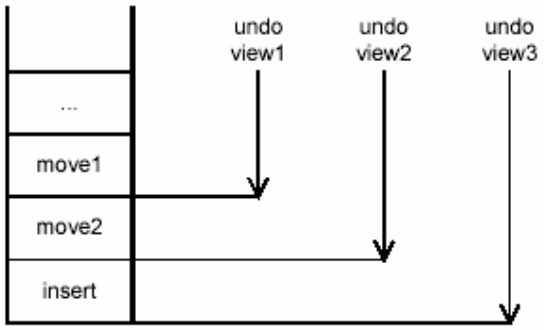

Fig. 43: Command history and multiple views

Nella precedente figura, lo stato della cronologia dei comandi è mostrato, dopo l'inserzione di una cella nel modello, il move nella seconda vista, e il conseguente move della prima vista. Dopo la inserzione della cella nel modello, la posizione della cella è la stessa in tutte le viste, ossia la posizione che è stata passata nella chiamata a **insert.** Le frecce illustrano gli edit da "undoare" con **GraphUndoManager** per le rispettive viste. Nel caso della vista 3, che vede solo l'inserzione, tutti gli edit sono "undoati".
Come menzionato precedentemente, anche se ci sono possibilmente molte sorgenti che notificano il **GraphUndoManager**, l'implemantazione di questo supporto per undo esiste in maniera unica, ossia nel modello del grafo. Perciò, il **GraphndoManager** deve solo essere aggiunto a un punto globale di ingresso (global entry point), che è l'oggetto **GraphModel**.

6 Conclusioni

Il componente Jgraph supporta opzioni estese di display e di editing, ma la semantica del grafo, ossia il significato di vertici e spigoli, è definito dalla applicazione. Perciò, Jgraph è altamente customizzabile, e le classi forniscono agganci (hooks) per le sottoclassi. Nei casi dove queste customizzazioni non sono sufficienti, il sorgente del componente deve essere disponibile per modificarne l'implementazione di default.

Per riassumere, i componenti per alberi e liste sono largamente usati per visualizzare strutture di dati, invece questo componente per grafi è di solito anche usato per modificare un grafo e per gestire queste modifiche in una maniera che dipende dall'applicazione.

A causa degli attributi di Jgraph, ciò che segue deve essere tenuto a mente :
Registrare una vista con il modello prima di usare il metodo **insert**, poiché altrimenti gli attributi che sono passati al metodo vengono ignorati, con il risultato di celle che hanno la stessa posizione e dimensione.

6.1 Celle e Componenti

A causa del setup speciale ereditato da Jtree, le celle e le viste delle celle di Jgraph non possono essere usate come componenti e viceversa, i componenti non possono essere aggiunti a Jgraph. Per usare viste di celle come un componente, un'istanza del suo renderer deve essere usata invece, fornendo una vista di cella dummy che permette al dato di essere visualizzato. Vice versa, per piazzare un componente in un grafo, il componente dovrebbe essere usato come un renderer per una specifica vista di cella.

6.2 Cambi Compositi

Usando l'intefaccia **GraphModel**, è solo possibile inserire, rimuovere o cambiare il modello, ma non è possibile creare ed eseguire cambiamenti compositi, che fanno più di una operazione alla volta. Per implementare cambiamenti compositi, un modello customizzato deve fornire metodi addizionali che o permettano di creare transazioni al di fuori di altre transazioni, o di accedere al modello in una manira orientata alla transazione, usando metodi **begin, commit e rollback.**

6.3 Scalabilità

Diversamente dal modello di default di Jtree che offre un flag che influenza il modo in cui i dati sono memorizzati e cachati, in Jgraph, un dato è sempre cachato dalle viste di cella.
Perciò, il modello di default è ben scalabile e deve solo essere esteso se l'algoritmo per la ricerca spaziale o per l'immagazinamento delle strutture di dati necessita di essere rimpiazzato con un algoritmo più efficiente.

7 Appendici

7.1 Model-View-Control (MVC)

Il termine MVC è di solito usato in congiunzione con l'architettura dei componenti della interfaccia dell'utente, e sta per Modello-Vista-Controllore, un design pattern che è spiegato in [17].
(Un design pattern è semplicemente " il cuore della soluzione a un problema che occorre più e più volte".)

In questo documento, il termine model e view sono usati nel senso del pattern MVC. Il termine modello è usato quando si parla del modello del grafo, ossia quando si parla di oggetti che implementano l'interfaccia **GraphModel**. Il termine view è usato nel contesto dell'oggetto **GraphView**, e per oggetti che implementano l'interfaccia **CellView**. In un certo senso, una cella può essere vista come un modello per la cellView, come il **GraphModel** è il modello per il grafo, e la **GraphView** è la vista per questo modello.

In generale, una vista non contiene mai i dati e un modello non contiene mai il codice di disegno. Questo è anche vero in Jgraph, con l'eccezione che affianco ai dati che il modello fornisce, la view contiene dati addizionali, che sono indipendenti dal modello.

Il pattern MVC è una collaborazione di design pattern, il più importante dei quali è il *Publish-Subscrive,* anche detto *Observer*, che definisce le relazioni tra il modello e la vista. In generale, il pattern *Observer* è usato per "definire una dipendenza uno a molti tra oggetti così che quando un oggetto cambia stato, tutte gli oggetti da lui dipendenti sono notificati e aggiornati automaticamente" [17].

Uno dei parametri del pattern Observer è come implementare lo schema di notifica, cioè, quanta informazione dovrebbe viaggiare tra il modello e la vista per descrivere un cambio. La soluzione di base calcola l'informazione sul cambio solo una volta nel modello e la dispaccia a tutti i listener, mentre un'altra soluzione semplicemente notifica i listener di un cambio e i listener invece usano il modello per calcolare le informazioni del cambio solo se sono interessati a gestire tale evento di modifica.

Ovviamente, la prima soluzione ha il vantaggio che l'informazione di cambio è computata una sola volta, mentre nella seconda soluzione, è computata per ogni vista attaccata. Lo svantaggio del primo approccio è che un sacco di informazione viaggia dal modello alla vista (anche su una rete), anche se la vista non è interssata a gestire l'evento. Certamente, queste soluzioni possono essere mixate in modo da computare parte della informazione di cambio nel modello e parte nei listeners. (Jgraph usa il primo approccio, così che ogni evento porta con se l'informazione completa sulla modifica, che viene anche usata per implementare la cronologia dei comandi.)

7.2 Swing

7.2.1 Swing MVC

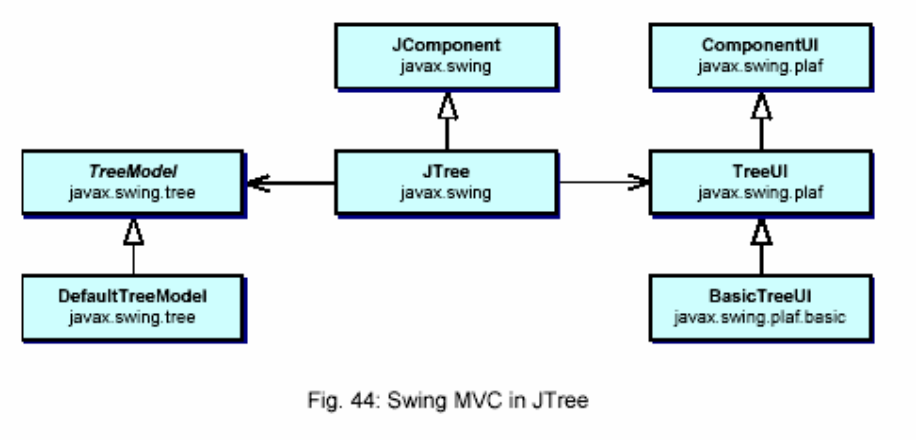

Fig. 44: Swing MVC in JTree

Jtree estende **Jcomponent** e mantiene un riferimento al suo **TreeUI**. Jtree ha un riferimento a **TreeModel** e istanzia **BasicTreeUI,** che estende **TreeUI,** che estende **ComponentUI** . **ComponentUI,** spesso chiamato il delegato UI, è in una gerarchia separata, di modo che possa essere scambiato indipendentemente quando cambia il L&F.

Questa struttura basilare, non triviale, dei componenti Swing è chiamata *Swing MVC Pattern.* Il controllore in Swing MVC è incapsulato nel delegato UI, che è responsabile del rendering del componente in una maniera dipendente dalla piattaforma, e del mappaggio degli eventi da interfaccia utente a transazioni eseguite dal modello.

Dal momento che Swing offre la possibilità di cambiare il L&F a run-time, il componente e il suo delegato UI sono implementati dentro differenti gerarchie, così che l'implementazione del delegato UI possa essere cambiata a run-time, indipendentemente dal componente. Il componente Jgraph eredita il setup di base dalla classe **Jcomponent** e dal suo delegato UI, che implementa la classe astratta **ComponentUI**. (Jgraph è responsabile di instanziare il corretto delegato UI quando cambia il L&F, fornendo un riferimento alla istanza corretta.)

7.2.2 MVC e il componente Text

In analogia alle caratteristiche dei componenti di text di Swing, che non cambiano quando il L&F cambia, le celle di Jgraph non dipendono dallo specifico L&F. Sebbene, lo schema dei colori, i collegamenti con la tastiera e altri aspetti del sistema dipendano dal L&F e devono rifletterlo. Come nel caso dei componenti text, la divisione tra attributi dipendenti e indipendenti dalla piattaforma è imlementata usando il concetto di views, che sono indipendenti dagli elementi che appaiono nel modello.
Nei componenti text di Swing, gli elementi del modello implementano l'interfaccia **Element**, e per ciascuno di questi elementi, esiste esattamente una vista. Queste viste, che implementano l'interfaccia **View**, sono o accedute attraverso un mappaggio tra gli elementi e le viste, o attraverso un punto di ingresso chiamato root view, che è referenziato dal delegato UI del componente text.

7.2.3 Jgraph MVC

Jgraph ha un settup analogo, con la sola differenza che una vista di grafo è referenziata dalla istanza stessa di Jgraph. Le celle del modello del grafo implementano l'interfaccia **GraphCell**, che è l'analogo Jgraph di **Element** e le viste di celle implementano l'interfaccia **CellView** analoga a **View**. La vista di cella è acceduto tramite il meccanismo di **CellMapper** o attraverso la graph view, che è una istanza di classe **GraphView**.

La classe **GraphView** può anche essere vista come una variazione del layout cachato di default da Jtree, poiché esso separa il modello dalla vista e fornisce informazioni di stato, dipendenti dalla vista per ogni cella, come la proprietà expanded di un tree node o uno dei suoi ancestori in Jtree. Poiché la classe **GraphView** lavora insieme con altre classi, l'analogia con il componente text di Swing è più utile per capire la separazione tra le celle e le loro viste ; anche se Jgraph è stato ottenuto dal codice di Jtree.

A differenza del componente text, dove gli attributi geometrici sono tenuti solo nel modello, Jgraph permette di conservarli separatamente in ciascuna vista, così da permettere al modello del grafo di avere più configurazioni geometriche, una per ogni vista attaccata. La classe **GraphView** può essere pertanto vista come una configurazione geometrica del modello dipendente dalla vista, che preserva lo stato quando cambia il L&F.

7.2.4 Serializzazione

La serializzazione è l'abilità di un oggetto di viaggiare attraverso una rete o di essere scritto su una memoria persistente. Due diversi tipi di serializzazione sono disponibili : a breve termine e a lungo termine. Quella breve è basata sul formato binario, l'altra invece su XML, rendendone il risultato leggibile dal'uomo.

Il componente Jgraph supporta la serializzazione a breve termine, mentre quella a lungo termine, implementata dalla JDK 1.4 non è ancora supportata. Perciò, la serializzazione non dovrebbe essere usata per implementare un formato di memorizzazione per l'applicazione. Ciò perché il formato potrebbe non essere compatibile con versioni future di Jgraph.

Non è in generale sufficiente conservare il solo modello, poiché parte delle informazioni, vedi il modello di layout del grafo, sono conservate nella vista (a meno che il modello non sia un attribute store). Pertanto, le viste dovrebbero essere memorizzate insieme al modello per mantenere il grafo con la sua corretta geometria.

7.2.5 Datatransfer

L'abilità di trasferire dati ad altre applicazioni o al sistema operativo per mezzo del Drag and Drop (DnD) o tramite gli *appunti* è sommariamente chiamato col termine datatransfer. DnD permette il trasferimento di dati usando il mouse, mentre gli appunti forniscono un buffer intermedio per conservare e recuperare i dati usando *cut, copy, paste*.

Tra la JDK 1.3 e 1.4, la funzionalità di datatransfer è stata estesa con un listener di alto livello, chiamato **TransferHandle**. Il **TransferHandle** è invocato dai listener di basso livello che sono già installati nella classe **Jcomponent**, e forniscono un set di metodi che unificano le funzionalità di DnD e degli appunti.

Ereditando da questa classe, il programmatore può specificare quali dati trasferire agli appunti o attraverso il DnD, e cosa accade dopo che un drop avviene con successo su un differente target, o sul componente Jgraph, ecome gestire contenuti con standard degli appunti.

Avendo detto ciò, il concetto di data flavors e di transferable devono essere spiegati.
Ciascun oggetto che è trasferito agli appunti implementa l'interfaccia **Transferable**, che fornisce metodi per recuperare le diverse forme nelle quali il transferable si rende disponibile alla applicazione (per esempio come HTML o testo normale). Il **Transferable** stesso può essere visto come un wrapper che contiene le differenti rappresentazioni (tipi MIME) del dato da trasferire, insieme con i metodi per accedere queste rappresentazioni in un modo type-safe.

Jgraph fornisce una implementazione di **Transferable** nella forma della classe **GraphTransferable,** che permette di trasferire la selezione corrente o come celle (oggetti serializzati) o come testo o come HTML. (Il **TransferHandler** di defalt di Swing non può essere usato, poiché trasferisce *proprietà bean*, e la selezione del grafo non può essere implementata come una *proprietà bean*.)

GraphTransferable trasferisce le celle serializzate insieme ai loro attributi, un connection set e una parent map che definiscono le relazioni esistenti nel modello tra le celle trasferite. Questi oggetti possono essere usati come argomenti nel metodo **insert** del modello target, o individualmente, per fornire un data flavor customizzato.

7.3 Packages

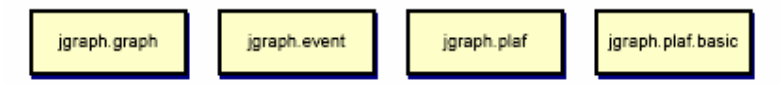

Fig. 45: JGraph's packages

I grafi sono costituiti da un certo numero di classi e interfacce definite nei loro propri package – il package **jgraph.graph**. Questo package fornisce classi di supporto che includono il modello del grafo, le celle del grafo, l'editor delle celle e i renderer. La classe **Jgraph** rappresenta il componente del grafo, che è la sola classe che risiede nel package **jgraph**.

Il package **jgraph.event** contiene le classe per gli eventi e i listener che sono usate per reagire agli eventi lanciati da **Jgraph**. La classe **GraphUI** nel package **jgraph.plaf** estende il componente swing **ComponentUI** e definisce i metodi astratti specifici dei grafi. La classe **BasicGraphUI** nel package **jgraph.plaf.basic** definisce l'implementazione di default per tali metodi.

In generale, la rappresentazione grafica dipende dal L&F, il che significa che il grafo ricarica la sua user interface (UI) quando li L&F cambia. La UI permette la selezione di celle singole o la selezione marquee, l'in-place editing, la modifica e lo spostamento delle celle. Zoom, cronologia dei comandi, DnD e appunti sono anche forniti.

Le celle hanno uno **user object**. Gli **user object** sono del tipo **Object** e perciò forniscono un modo per associare qualsiasi tipo di oggetto ad una cella. I grafi hanno un modello semplice e ciascuna istanza di **Jgraph** mantiene un riferimento al modello del grafo.
Le classi chiave del package **jgraph.graph** sono mostrate di seguito.

`DefaultGraphModel`	A graph model that defines methods for adding, removing and changing cells.
`GraphView`	An object that constitutes the geometric pattern of the graph.
`DefaultGraphCell`	The base class for all elements in the model, also represents a vertex.
`DefaultEdge`	An edge that may connect two vertices via a source and target port.
`DefaultPort`	Acts as a child of a vertex, and the connection point for edges.

Table 2: Key classes from the jgraph.graph package

7.4 Class Index

GraphCell (see 3.2.1)

GraphCellEditor (see 5.3)

GraphConstants (see 2.5.1)

GraphContext (see 4.6)

GraphModel (see 3.3.1)

GraphSelectionModel (see 3.5)

GraphTransferable (see 5.5)

GraphUndoManager (see 5.7.4)

GraphView (see 4.1)

ParentMap (see 3.4.3)

Port (see 3.2.1)

PortRenderer (see 5.2.2)

PortView (see 4.4.2)

VertexRenderer (see 5.2.2)
VertexView (see 4.4.2)

Plaf

GraphUI (see 5.1.1)

Basic

BasicGraphDropTargetListener (see [19])

BasicGraphUI (see 5.1.2)

BasicTransferable (see [19])

7.5 UML reference

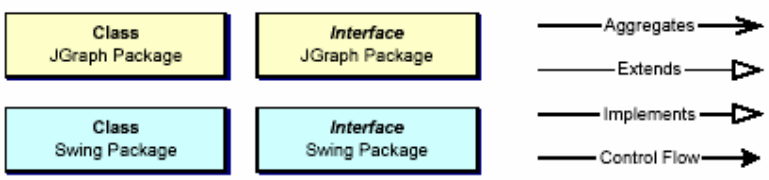

Fig. 46: UML for static structure diagrams

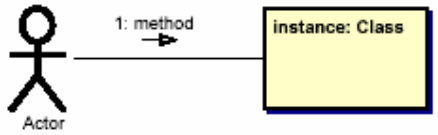

Fig. 47: UML for collaboration diagrams

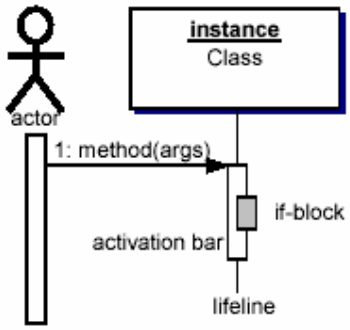

Fig. 48: UML for sequence diagrams

8 References

[1] Biggs.Discrete Mathematics (Revised Edition).Oxford University Press,New York NY,1989.
[2] Aigner.Diskrete Mathematik.Vieweg,Braunschweig/Wiesbaden, 1993.
[3] Ottmann,Widmayer.Algorithmen und Datenstrukturen (2.Auflage). BI-Wiss.-Verl.,Mannheim,Leipzig,Wien,Zürich.1993
[4] Sedgewick.Algorithmen.Addison-Wesley,Reading MA,1991.
[5] Nievergelt,Hinrichs.Algorithms and data structures.Prentice-Hall, Englewood Cliffs NJ,1993.
[6] Harel.Algorithmics.Addison-Wesley,Reading MA,1992.
[7] Bishop.Java Gently.Addison-Wesley,Reading MA,1998.
[8] Jackson,McClellan.Java 1.2 by example.Sun Microsystems,Palo Alto CA,1999.
[9] Flanagan.Java in a nutshell (2nd Edition).O'Really & Associates, Sebastopol CA,1997.
[10] Stärk,Schmid,Börger.Java and the Java Virtual Machine. Springer,Heidelberg,2001.
[11] Geary.Graphic Java 2,Volume II: Swing (3rd Edition).Sun Microsystems,Palo Alto CA,1999.
[12] Geary.Graphic Java 2,Volume III: Advanced Swing (3rd Edition). Sun Microsystems,Palo Alto CA,1999 (?).
[13] The Swing Tutorial.
http://java.sun.com/docs/books/tutorial/index.html
[14] JTree API Specification.
http://java.sun.com/j2se/1.4/docs/api/javax/swing/JTree.html
[15] Violett.The Element Interface.
http://java.sun.com/products/jfc/tsc/articles/text/element_interface/
[16] View Interface API Specification.
http://java.sun.com/j2se/1.4/docs/api/javax/swing/text/View.html
[17] Gamma,Helm,Johnson,Vlissides.Design Patterns.Addison-Wesley,Reading MA,1995.
[18] Alder.The JGraph Tutorial.
http://jgraph.sourceforge.net/tutorial.html
[19] The Java 1.4 API Specification.
http://java.sun.com/j2se/1.4/docs/api/
[20] Alder.The JGraph 1.0 API Specification.http://api.jgraph.com
[21] Alder.JGraph.Semester Work,Department of Computer Science, ETH Zürich,Switzerland,2001.