Visualizzatore Piani SQL

Specifica Tecnica

Informazioni documento

Titolo documento:	Specifica Tecnica
Data creazione:	2011/10/25
Versione attuale:	1.0.0
Stato del documento:	Formale a uso esterno
Nome file:	Specifica_tecnica_1.0.0.pdf
Redazione:	Luca Fongaro

DATA	VERSIONE	REDATTORI	MOTIVO DELLA MODIFICA
2011/10/20	1.0.0	Luca Fongaro	RILASCIO VERSIONE Rilascio della versione 1.0
2011/10/19	0.1.0	Luca Fongaro	PRIMA STESURA. Prima stesura del documento

Indice

Elenco delle figure

Elenco delle tabelle

1 Introduzione

1.1 Scopo del documento

Il presente documento ha lo scopo di definire l'architettura dell'interfaccia per il tool *Visual Query*. Tale definizione inizia descrivendo il prodotto ad alto livello, dopo la quale segue un'analisi a basso livello.

1.2 Scopo del prodotto

Il visualizzatore di piani SQL ha come obiettivo il fornire una visione semplice e completa di un piano di esecuzione SQL. Creando quest'interfaccia sarà possibile avere una visione più chiara di come una *query* verrà eseguita da un determinato database potendo quindi ottimizzarla nel caso si accerti la presenza di operazione onerose o da evitare. L'interfaccia sarà parte integrante del tool *Visual Query* che è una componente del prodotto software *SITEPAINTER Portal Studio*, di proprietà dell'azienda *Zucchetti SpA*.

1.3 Riferimenti

- JavaScript Documentation https://developer.mozilla.org/en/JavaScript
- Raphael Reference http://raphaeljs.com/reference.html
- Raphael ZPD Reference https://github.com/semiaddict/raphael-zpd
- XHTML Reference http://www.w3.org/TR/2002/REC-xhtml1-20020801/
- HTML5 http://www.w3.org/TR/html5/

2 Definizione del prodotto

2.1 Architettura di SITEPAINTER Portal Studio

Il prodotto si basa sull'architettura *Three Tier*, la quale viene spesso considerata come un evoluzione del modello *Client-Server* in quanto viene aggiunto un nuovo modulo che gestisce il trasferimento dei dati dal livello *client* al livello dell'archivio dei dati. Questo modulo aggiuntivo rende indipendente il livello *client* dai problemi di accesso al livello *server*. Il pattern architetturale *Three Tier* divide quindi il sistema in tre diversi moduli, dedicati all'interfaccia utente, alla logica funzionale (*business logic*) e alla gestione dei dati persistenti. Questi tre livelli sono generalmente identificati in:

- Un'interfaccia, spesso rappresentata da un *Web Server* e da eventuali contenuti statici;
- La *Business logic*, che in genere coincide con un application server che genera i contenuti dinamici;
- I dati memorizzati, che sono manipolati dalla *Business Logic* sono memorizzati quasi sempre in un database, che ne assicura la persistenza.

L'ambiente di sviluppo aziendale, SitePainter Infinity, si basa proprio su questa struttura e si compone quindi di 3 strati:

- *Presentation Tier*
- *Business Tier*
- *Data Tier*

Per capire meglio come funzionano i tre strati verrà ora analizzato il percorso con cui vengono processate le richieste dell'utente. Inizialmente l'utente digita sulla barra degli indirizzi del proprio *browser* un *URL*, la richiesta viene quindi trasformata in una operazione di *POST* o *GET*, che viene poi indirizzata al *server Web*. A questo punto le richieste statiche vengono fornite dal *server Web* stesso, mentre quelle dinamiche vengono inoltrate all'*Application Server* dove saranno processate da una applicazione (secondo strato) che creerà dei *Business Object* che rappresenteranno le richieste. In caso di necessità di lettura dei dati, vengono inviate le richieste al *Database Server*, ovvero al terzo ed ultimo strato. Il database processa quindi la richiesta e ritorna i dati al *Business Object*, con i quali viene creata una pagina HTML di risposta. Tale pagina viene passata al *Web Server* e infine torna indietro fino al *browser* dell'utente. Solitamente *Web Server* e *Application Server* risiedono nella stessa macchina, in quanto hanno un gran volume di dati da scambiare, mentre il *Business Logic* può risiedere sulla stessa macchina del *Web Server*, in caso di piccole applicazioni (si pensi ai server così detti *LAMP*), o, in alternativa, su un *cluster* composto da molti computer nel caso di applicazioni utilizzate da migliaia di utenti: in questo caso il Web Server inoltra i parametri necessari verso la macchina destinata a rispondere all'utente in questione. La figura sottostante (figura 1 a pagina 3) è a puro scopo illustrativo e non è una rappresentazione corretta e completa del funzionamento di SITEPAINTER, ma aiuta a capirne il funzionamento di base.

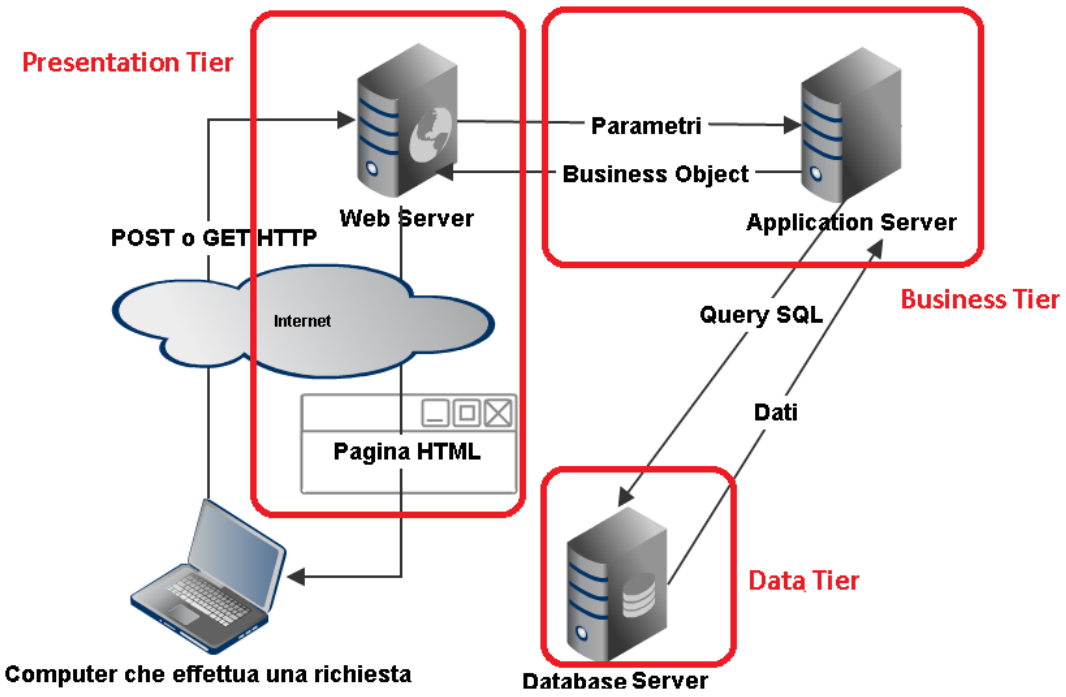

Figura 1: Schema funzionamento SITEPAINTER

2.2 Struttura della componente Visual Query

La struttura di una componente integrabile con *SITEPAINTER Portal Studio* è formata generalmente da tre parti:

- Il file *JavaScript*, che deve chiamarsi *nomeComponenteObj.js*, rappresenta il nucleo della componente stessa;
- Il file *.edtdef* che definisce le proprietà personalizzabili della componente, l'inizializzazione dell'oggetto e altre proprietà che riguardano la componente;
- Il file *java*, che deve chiamarsi *nomeComponenteControl.java* che si occuperà di generare la pagina *jsp*, associando i vari file *.js*.

Ad ogni file *js* o *java* possono essere associati ulteriori file *js* o *java* In questo stage saranno sviluppati esclusivamente dei file *js* e una pagina *jsp*, integrandosi con un tool già esistente, per cui lo studente non dovrà creare i file sopra menzionati.

3 Tecnologie e librerie utilizzate

3.1 HTML 5

L'ultima versione in sviluppo del linguaggio HTML. Anche se è classificata come *W3C Working Draft*, ovvero significa che le specifiche potranno essere soggette a modifiche in futuro, si può considerare abbastanza stabile e i motori di *rendering* dei principali *browser* offrono un supporto abbastanza ampio ai *tag* più diffusi del linguaggio stesso.

3.2 JavaScript

È un linguaggio di *scripting* orientato agli oggetti, scarsamente tipizzato e standard *ECMA*. Essendo un linguaggio di *scripting* non viene compilato, ma interpretato dal motore di *rendering* del *browser*. La conseguenza a questo comportamento è un importante vantaggio, ovvero il server non deve eseguire del codice, ma solo inviarlo e sarà il *client* ad eseguirlo. Il linguaggio presenta anche due importanti svantaggi:

- Il codice *javascript* va completamente scaricato prima di poter essere eseguito, questo può essere un problema se la connessione ad internet risulta lenta, ma l'ormai diffusa banda larga, anche in Italia, riduce, se non elimina, questo problema;
- Essendo l'intero codice a disposizione del *client* stesso, questo può porre dei problemi di sicurezza in quanto può essere alterato il codice stesso ed esporre il server all'invio di dati non corretti, costringendo a scrivere ulteriore codice di controllo lato server. Nel caso specifico questo problema non esiste, in quanto l'interfaccia sviluppata in questo progetto non invia dati al server, ma li riceve solamente.

Un'altra caratteristica di questo linguaggio è il considerare tutto come un oggetto, per questo non esistono classi, ma esclusivamente oggetti. Anche la definizione di un'ipotetica classe è un oggetto e gli oggetti che utilizzano questa definizione non sono delle istanze, come lo sarebbero in linguaggi come Java, ma degli altri oggetti che utilizzano la definizione come prototipo.

3.3 Raphael

Una piccola libreria scritta in JavaScript da Dmitry Baranovskiy che permette la creazione di oggetti grafici utilizzando le SVG W3C Recommendation o VML. Ogni oggetto creato con questa libreria è anche un oggetto *DOM* e questo offre la possibilità di associare ad esso degli eventi, permettendo quindi due cose: la creazione in modo facile di oggetti grafici e la loro manipolazione in modo altrettanto semplice.

3.4 RaphaelZPD

Questa piccola libreria permette di estendere Raphael affinché sia possibile utilizzare le funzioni di *zoom, drag e pan*. In essa vi è presente un bug che non permette la corretta associazione tra l'evento che attiva lo *zoom* e la funzione che lo gestisce. Il bug riguarda i browser Internet Explorer ed Opera. In questi browser la libreria anziché associare l'evento *mousewheel*, correttamente supportato, viene associato l'evento *DOMMouseScroll* che invece è supportato solo dal browser Firefox. Entrambi gli eventi si riferiscono al movimento della rotellina del mouse. Un membro interno dell'azienda avendo riscontrato questo bug in un'occasione precedente ha già provveduto a correggerlo, modificando la libreria originale. Il codice originale affetto dal bug era:

```
if (navigator.userAgent.toLowerCase().indexOf('webkit') > = 0) {
   me.root.addEventListener('mousewheel', me.handleMouseWheel, false);
// Chrome/Safari }
else {
   me.root.addEventListener('DOMMouseScroll', me.handleMouseWheel, fal-
se);
// Gli Altri}
```

La correzione è stata la seguente:

```
me.root.addEventListener('mousewheel', me.handleMouseWheel, false);
//Tutti gli altri
me.root.addEventListener('DOMMouseScroll', me.handleMouseWheel, false);
// Firefox
```

Ora la libreria associa entrambi gli eventi, ovvero *mousewheel* e *DOMMouseScroll* ad un qualsiasi browser, ma questo non crea alcun problema. In seguito all'analisi della libreria lo studente ha inoltre apportato un'altra piccola modifica, che permette di escludere che determinati oggetti grafici attivano la funzionalità di *pan* (di default attivata da un click del mouse, ovvero l'evento *onMouseDown*, in un qualunque punto dell'area disegnata). La modifica è ispirata già da quanto ha fatto l'autore per disabilitare la funzionalità *drag* su specifici elementi, ovvero vi è un controllo su un campo dell'oggetto coinvolto nell'attivazione dell'attività di *drag*. Il campo controllato è *evt.target.draggable*, dove *evt* indica un evento e target indica l'oggetto che è coinvolto nell'evento. Basandosi su questo è stata modificata la seguente riga di codice:

```
if (!me.opts.pan) return;
```

in

```
if (!me.opts.pan || evt.target.canPan == false) return;
```

Se il campo *canPan* è false esegue un *return* e non l'azione di *pan*. Se un oggetto non ha questo campo non viene provocato alcun errore. La modifiche di questa libreria non rappresentano un problema, essendo non più sviluppata, questo perché in *Raphael* sono in sviluppo le stesse funzionalità. Nella versione più recente (2.0.1) esse sono implementate, ma solo il *drag* è efficiente, mentre lo *zoom* e il *pan* appesantiscono di molto l'esecuzione del codice da parte del browser. Quindi, solo per queste due ultime funzionalità la libreria sarà utilizzata e verrà indicata da qui in avanti come *RaphaelZPDmodify*, per segnalare che non è la versione originale.

4 Specifica delle componenti

Nel seguente diagramma delle classi *UML* sono illustrate le relazioni tra le singole classi. In seguito ogni classe sarà descritta ad alto livello, mentre solo le classi più rilevanti saranno descritte in modo più approfondita. Le classi Raphael e RaphaelZPDModify sono le librerie esterne e non saranno qui illustrate. Infine verrà illustrato brevemente l'algoritmo utilizzato per la creazione di un albero, comune a tutte le modalità possibili.

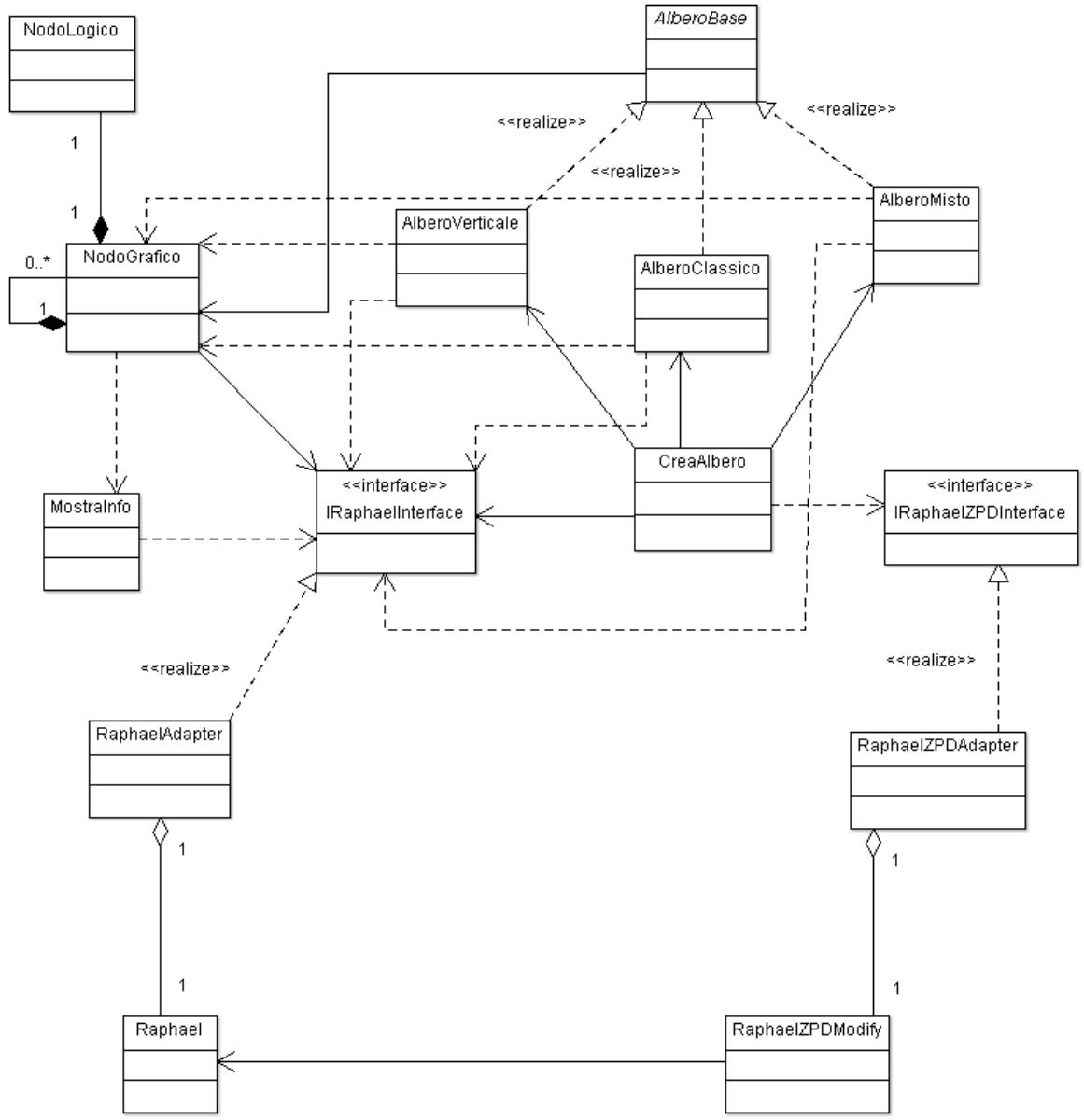

Figura 2: Diagramma Componenti

4.1 Descrizione delle componenti ad alto livello

4.1.1 Nodo Logico

Tipo : Class;

Nome : NodoLogico;

Descrizione : Rappresenta la parte logica di un'operazione di una particolare *query*, contenendo le informazioni relative al nome, il colore (giudizio) e le altre informazioni aggiuntive;

Relazioni d'uso o d'interfaccia con altri componenti : Nessuna

4.1.2 Mostra Info

Tipo : Class;

Nome : MostraInfo;

Descrizione : Si occupa di visualizzare tutte le informazioni aggiuntive di una singola operazione;

Relazioni d'uso o d'interfaccia con altri componenti : Utilizza un riferimento di tipo IRaphaelInterface per creare degli oggetti grafici

4.1.3 Nodo Grafico

Tipo : Class;

Nome : NodoGrafico;

Descrizione : Rappresenta la parte grafica di un'operazione di una particolare *query*, illustrando le informazioni dell'operazione stessa e permettendo di effettuare sul sottoalbero il *drag*, di nasconderlo e di evidenziarlo, ovvero nascondere il resto;

Relazioni d'uso o d'interfaccia con altri componenti : Possiede un riferimento ad un NodoLogico per memorizzare i dati associati all'operazione, riferimento al NodoGrafico padre, se non è la radice, e ai Nodografico figli, se presenti. Possiede inoltre un riferimento di tipo IRaphaelInterface per disegnare i singoli oggetti grafici. Utilizza la classe MostraInfo per mostrare a video le informazioni aggiuntive di un'operazione.

4.1.4 IRaphaelInterface

Tipo : Interface;

Nome : IRaphaelInterface;

Descrizione : Un'interfaccia che definisce i metodi utilizzabili dagli altri componenti per creare gli oggetti grafici. In questo ambito tecnologico l'interfaccia sarà fittizia in quanto JavaScript non permette la creazione di questo costrutto;

Relazioni d'uso o d'interfaccia con altri componenti : Nessuno.

4.1.5 RaphaelAdapter

Tipo : Class;

Nome : RaphaelAdapter;

Descrizione : Classe che richiama i metodi della libreria Raphael per ottenere degli oggetti grafici;

Relazioni di implementazione o ereditarietà : Implementa l'interfaccia IRaphaelInterface.

Relazioni d'uso o d'interfaccia con altri componenti : Mantiene un riferimento all'oggetto di tipo *Raphael*, ottenuto dalla libreria Raphael.

4.1.6 IRaphaelZPDInterface

Tipo : Interface;

Nome : IRaphaelZPDInterface;

Descrizione : Un'interfaccia che definisce un metodo per utilizzare la libreria RaphaelZPDModify. In questo ambito tecnologico l'interfaccia sarà fittizia in quanto JavaScript non permette la creazione di questo costrutto;

Relazioni d'uso o d'interfaccia con altri componenti : Nessuno.

4.1.7 RaphaelZPDAdapter

Tipo : Class;

Nome : RaphaelZPDAdapter;

Descrizione : Classe che richiama il metodo della libreria RaphaelZPDModify per attivare su un oggetto Raphael le funzionalità offerte dalla libreria;

Relazioni d'implementazione o ereditarietà : Implementa l'interfaccia IRaphaelZPDInterface;

Relazioni d'uso o d'interfaccia con altri componenti : Nessuno.

4.1.8 AlberoBase

Tipo : AbstractClass;

Nome : AlberoBase;

Descrizione : Offre le funzionalità base comuni a tutte le modalità di disegno dell'albero;

Relazioni d'uso o d'interfaccia con altri componenti : Nessuno.

4.1.9 Albero Misto

Tipo : Class;

Nome : AlberoMisto;

Descrizione : Si occupa di creare i nodi grafici e di ordinarli in modo da creare un albero diviso in due, sinistra e destra. La parte sinistra ha queste caratteristiche:

- Ogni nodo di un certo livello è allineato con qualunque nodo dello stesso livello;
- Ogni nodo padre ha sotto di sé, se ne possiede, i nodi figli, ma non è centrato rispetto ad essi.

La parte destra è speculare, ma i nodi anziché allineati sono incolonnati e i nodi sono a destra del padre e non sotto.

Relazioni d'implementazione o ereditarietà : Eredita da AlberoBase;

Relazioni d'uso o d'interfaccia con altri componenti : crea gli oggetti di tipo NodoGrafico e passa ad essi un riferimento a RaphaelAdapter.

4.1.10 Albero Verticale

Tipo : Class;

Nome : AlberoVerticale;

Descrizione : Si occupa di creare i nodi grafici e di ordinarli in modo da creare un albero diviso con queste caratteristiche:

- Ogni nodo di un certo livello è incolonnato rispetto ai nodi dello stesso livello;
- Ogni nodo è distanziato in altezza dal fratello successivo in base alla grandezza del proprio sottoalbero;
- Il primo figlio di un certo nodo è spostato rispetto al padre a destra e in basso in base della grandezza del nodo padre.

Relazioni d'implementazione o ereditarietà : Eredita da AlberoBase;

Relazioni d'uso o d'interfaccia con altri componenti : crea gli oggetti di tipo NodoGrafico e passa ad essi un riferimento a RaphaelAdapter.

4.1.11 Albero Classico

Tipo : Class;

Nome : AlberoClassico;

Descrizione : Si occupa di creare i nodi grafici e di ordinarli in modo da creare un albero diviso con queste caratteristiche:

- Ogni nodo di un certo livello è allineato con qualunque nodo dello stesso livello;
- Ogni nodo padre è incolonnato in modo tale che l'ipotetica linea che dividerebbe a metà il sottoalbero sia un'asse di simmetria per il nodo padre stesso.

Relazioni d'implementazione o ereditarietà : Eredita da AlberoBase;

Relazioni d'uso o d'interfaccia con altri componenti : crea gli oggetti di tipo NodoGrafico e passa ad essi un riferimento a RaphaelAdapter.

4.1.12 CreaAlbero

Tipo : Class;

Nome : CreaAlbero;

Descrizione : Classe che si occupa di memorizzare il JSON ottenuto dal server, di creare l'oggetto adatto a disegnare l'albero in base alla scelta effettuata dall'utente e di attivare le funzionalità di *pan* e *zoom*.

Relazioni d'uso o d'interfaccia con altri componenti : Possiede un riferimento di tipo AlberoBase che verrà istanziato al sottotipo corretto in base alla scelta dell'utente, crea e mantiene un riferimento a *IRaphaelInterface* per creare la superficie da disegno e crea un riferimento a IRaphaeelZPDInterface per implementare la funzionalità di *zoom* e *pan*.

4.2 Descrizione in dettaglio delle singole componenti

Ora verranno descritte in modo più dettagliato alcune delle classi illustrate. I metodi con cinque o più parametri non sono rappresentati all'interno del diagramma e sono contrassegnati da un asterisco (*) per mantenere la leggibilità. Verranno omessi nella descrizione la spiegazione dei metodi *get* e *set* il cui comportamento è deducibile dal nome dell'operazione stessa. Il segno -(minore) indicherà le funzioni private, o meglio quelle operazioni che nel linguaggio vero e proprio non saranno dichiarate come *this.function()*, risultando inaccessibili dall'esterno della classe. Ogni membro di una classe è contraddistinto dal carattere *underscore* ('_'), con l'eccezione dei membri che rappresenterebbero delle variabili statiche.

4.2.1 NodoLogico

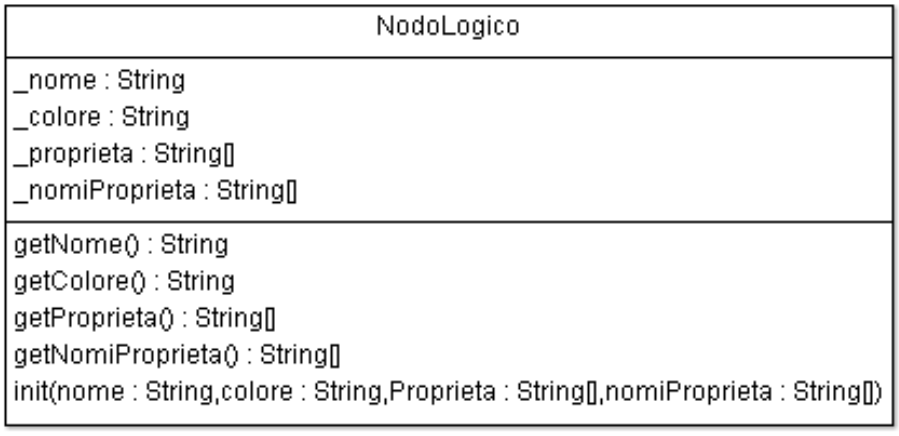

Figura 3: Diagramma di classe Nodo Logico

4.2.1.1 Membri

- _nome: contiene una stringa che contiene il nome di un'operazione;
- _colore: contiene una sa stringa che contiene il colore che indica il giudizio dato all'operazione lato server;

- _proprieta: contiene un array che memorizza tutti i valori delle altre informazioni dell'operazione, più il valore del nome dell'operazione;
- _nomiProprieta: contiene un array che memorizza tutti i nomi delle informazioni dell'operazione.

4.2.1.2 Funzionalità

- init(nome:String, colore:String, proprieta:String[], nomiProprieta:String[]): costruttore dell'oggetto. Controlla che i valori siano coerenti, trasforma il valore di colore da letterale (es.: red) a una stringa che codifica il valore nello standard HEX (es.: #ff9966).

4.2.2 NodoGrafico

Figura 4: Diagramma di classe Nodo Grafico

4.2.2.1 Membri

- _contenitore: rettangolo che delimita visualmente un NodoGrafico;
- _figli: può essere vuoto, array che contiene i riferimenti ai figli;
- _padre: null se è la radice, contiene il riferimento al NodoGrafico che funge da padre;
- _connettore: rappresenta una linea che parte dal figlio e arriva al padre;
- _infoLogiche: riferimento a NodoLogico che mantiene le varie informazioni logiche;
- _hideButton: immagine a cui è associata la funzionalità per nascondere il proprio sottoalbero;
- _hasHideSubTree: valore booleano per indicare se il nodo ha il proprio sottoalbero nascosto o meno;
- _canvas: riferimento a RaphaelAdapter utilizzato per creare i vari oggetti grafici richiesti;
- _iconReDraw: immagine a cui è associato l'evento che permette di considerare solo il sottoalbero, ovvero nascondere il resto dell'albero;
- _isReDraw: valore booleano per indicare se il nodo ha il padre nascosto.

4.2.2.2 Funzionalità

- **init(nome: String, colore: String, infoAggiuntive: String[], nomiInfoAggiuntive: String[], x: Integer, y:Integer, larghezza:Integer, altezza:Integer, canvas:RaphaelAdapter, padre:NodoGrafico)**: costruttore, i primi quattro parametri identificano le informazioni logiche del nodo, i quattro successivi la dimensione e le coordinate nello spazio, infine canvas è il riferimento utilizzato per disegnare gli oggetti grafici richiesti e padre è il riferimento al padre, se non è la radice, del nodo grafico. Il costruttore si occupa di creare la struttura base del nodo, ovvero di mostrare a video il nome, il colore associato e la possibilità di visionare le informazioni aggiuntive. In seguito se il nodo ha un padre aggiungerà le funzioni di *drag* sul proprio sottoalbero e per poter considerare il proprio sottoalbero, ovvero nascondere il resto dell'albero;
- **getFirstAncientRedraw()**: ritorna il primo antenato di un nodo il cui membro *_iconReDraw* è uguale a true;
- **getLarghezzaSubTree()**: ritorna la larghezza del sottoalbero di un certo nodo, se invocato sulla radice ritorna la larghezza dell'albero;
- **getAltezzaSubTree()**: ritorna l'altezza del sottoalbero di un certo nodo, se invocato sulla radice ritorna l'altezza dell'albero;
- **hasHiddenChild()**: ritorna true se l'intero sottoalbero del nodo, escluso il nodo stesso, è nascosto;
- **getMinWidthSubTree()**: ritorna il valore minore occupato dall'albero sull'ipotetico asse x;
- **getMinHeightSubTree()**: ritorna il valore minore occupato dall'albero sull'ipotetico asse y;
- **spostaFigli(dx: Integer, dy: Integer)**: sposta i figli dei valori specificati come parametri.

- **aggiornaConnettore()**: aggiorna il connettore che va dal figlio al padre;
- **associarDag()**: attiva la funzionalità di *drag*;
- **addFiglio(figlio: NodoGrafico)**: associa al nodo un figlio, se è il primo figlio associato aggiunge la funzionalità per nascondere il proprio sottoalbero;
- **deleteTree()**: richiama sulla radice la funzione deleteSubTree(), indipendentemente dal nodo su cui viene invocato;
- **deleteSubTree()**: cancella il sottoalbero del nodo;
- **hideShowFigli(nascondiMostra: boolean)**: richiamando la funzione hideshow ricorsivamente, nasconde o mostra il sottoalbero del nodo da cui è stata richiamata;
- **hideShow(nascondiMostra: boolean, nascondiMostraConnettore: boolean)**: nasconda o mostra il nodo su cui è stata richiamata in base al valore di nascondiMostra. Se il nodo non è la radice valuta anche il secondo parametro per decidere se nascondere o mostrare il connettore verso il padre;
- **controllaSpazio(figlio: NodoGrafico, posto: String)**: ritorna true se in una certa direzione, rispetto a figlio vi è un fratello sinistro del figlio stesso;
- **calcoloCoordinate(x:Integer, y:Integer, a:Integer, b:Integer, altezzaXY:Integer, altezzaAB:Integer, larghezzaXY:Integer, larghezzaAB:Integer)** in base alle coordinate e alle dimensioni di due rettangoli, restituisce un array dove sono memorizzati i valori ideali da dove parte e finisce il connettore tra i due. L'array restituito è composto da 4 valori, primi due rappresentano la coordinata y e x del punto dove il connettore si collega al padre, gli altri due valori il punto in cui il connettore parte dal figlio;
- **ottieniCoordinate()**: restituisce un array con le coordinate per disegnare una curva in modo tale che parta da un certo punto, passi per due punti stabiliti e quindi termini in un altro punto. L'array è composto da due array di interi, il primo contiene le coordinate x e y dei punti di inizio e fine della curva, il secondo le coordinate dei punti in cui la curva deve passare;
- **reDrawFunction(outerThis: NodoGrafico)**: funzione che nasconde tutto tranne il sottoalbero del nodo passato come parametro. L'uso di outerThis è necessario in quanto all'interno di questa funzione ne viene dichiarata un'altra, questo implica un cambiamento all'interno di quest'ultima dello scope e vi è la necessità di riferirsi al nodo;
- **riMostraSottoAlberoPrecedente(outerThis: NodoGrafico)**: viene invocata solo da un nodo che non sia la radice e il cui padre è nascosto. La funzione mostra il sottoalbero precedente su cui era stata chiamata *reDrawFunction()* oppure l'albero intero

4.2.3 MostraInfo

Figura 5: Diagramma di classe di MostraInfo

4.2.3.1 Membri

- _rettangoloContenitore: rettangolo che delimita il contenuto dell'oggetto;
- _rettangoloRilevaEvento: rettangolo delle stesse dimensioni di *_rettangoloContenitore* che si occupa di rilevare l'evento *onmouseout*;
- _arrayText: contiene gli elementi testuali disegnati;
- _arrayPath: contiene le linee disegnate per separare le singole informazioni, utili per aiutare una persona a leggere meglio il testo;
- _isDisegnato: questo valore è *true* se l'oggetto è visibile a video

4.2.3.2 Funzionalità

- **crea(nomiProp: String[], infoProp: String[], canvas:RaphaelAdapter, x:Integer, y:Integer)**: disegna a video i valori di nomiProp e infoProp, partendo dalla posizione (x,y);
- **remove()**: cancella quanto disegnato;

4.2.4 IRaphaelInterface

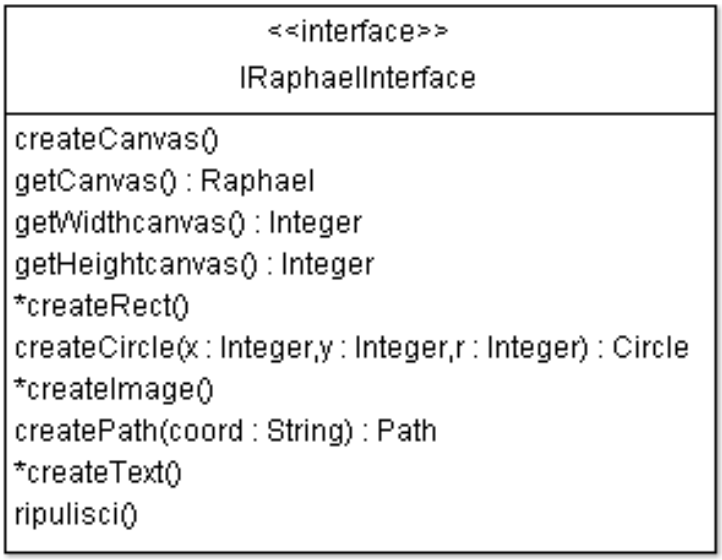

Figura 6: Diagramma di classe di IRaphaelInterface

4.2.4.1 Funzionalità

- **createCanvas()**: crea un oggetto di tipo Raphael che rappresenta la superficie dove saranno disegnati i componenti di dimensione (0,0). Se è già istanziato non fa nulla;
- **createRect(x:Integer, y:Integer, larghezza:Integer, altezza:Integer, angoliSmussati:Integer)**: crea un oggetto di tipo *Rect*, tipo proprio della libreria Raphael, utilizzando i primi quattro parametri come dimensione, *angoliSmussati* indica quanto gli angoli devono essere arrotondati;
- **createCircle(x:Integer, y:Integer, r:Integer)**: crea un oggetto di tipo *Circle*, tipo proprio della libreria Raphael, utilizzando i primi 2 parametri come posizione, il terzo come raggio;
- **createImage(path:String, x:Integer, y:Integer, larghezza:Integer, altezza:Integer)**: crea un oggetto di tipo *Image*, tipo proprio della libreria Raphael, dove *path* è il percorso dove risiede l'immagine, gli altri i parametri di dimensione;
- createPath(coord: String) return Path: crea un oggetto di tipo path, che rappresenta una linea che può essere costruita in diversi modi. *coord* dev'essere una stringa che rispetta le direttive SVG per i path. Le direttive sono consultabili a questo indirizzo web: http://www.w3.org\/TR\/SVG\/paths.html#PathData

4.2.5 IRaphaelZPDInterface

Figura 7: Diagramma di classe di IRaphaelZPDInterface

4.2.5.1 Funzionalità

- init(canvas: Raphael, isZoom:boolean, isPan:boolean, isDrag:boolean): permette di aggiungere l3 funzionalità zoom, pan o drag all'oggetto *canvas* passato, che dev'essere di tipo Raphael.

4.2.6 CreaAlbero

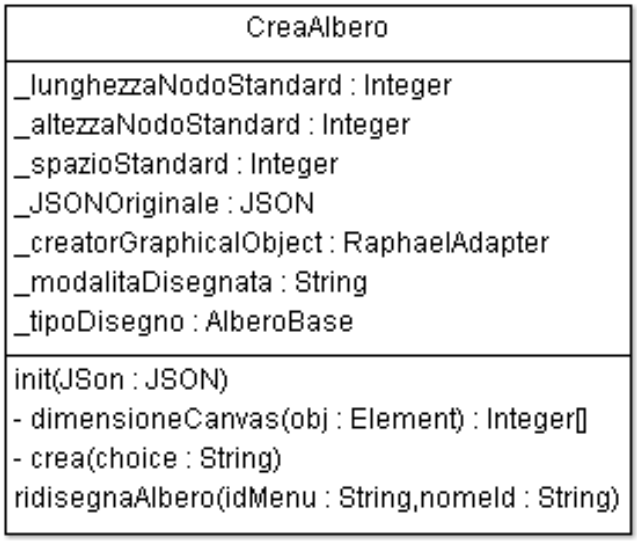

Figura 8: Diagramma di classe CreaAlbero

4.2.6.1 Membri

- _lunghezzaNodoStandard: lunghezza che dovranno avere i vari nodi creati;
- _altezzaNodoStandard: altezza che dovranno avere i vari nodi creati;
- _spazioStandard: spazio minimo tra un nodo e un altro;
- _JSONOriginale: JSON ottenuto dal server;
- _creatorGraphicalObject: riferimento a RaphaelAdapter;
- _modalitaDisegnata: identifica l'ultima modalità che è stata creata;
- _tipoDisegno: riferimento ad un sottotipo di AlberoBase;

4.2.6.2 Funzionalità

- **init(JSon:JSON, nomeId:String)**: questa funzione inizializza la variabile *_JSONOriginale*;
- **dimensioneCanvas(obj: Element)**: in base all'oggetto DOM passato, restituisce le dimensioni che questo oggetto può occupare in base alla dimensioni della finestra e di altri oggetti che eventualmente lo contengono. In questo ambito viene utilizzato per calcolare la dimensione che dovrà occupare il div che contiene il canvas. Restituisce un array con le dimensioni che dovrà occupare il canvas stesso;

- **crea(choice: String)**: il metodo si occupa di inizializzare *_tipoDisegno* in base al valore di *choice*, stringa che rappresenta la scelta dell'utente e quindi richiamare il metodo per la creazione dell'albero;
- **ridisegnaAlbero(idMenu: String, nomeId: String)**: il metodo recupera da un oggetto DOM il cui Id è il valore di idMenu la scelta dell'utente, se è uguale a quella già disegnata non fa nulla. Se è diversa cancella quanto disegnato e crea un oggetto di tipo RaphaelAdapter passando al costruttore il valore di nomeId, ovvero il nome dell'Id dell'oggetto DOM dove sarà disegnato il grafico. Richiama infine il metodo *crea*.

4.2.7 AlberoBase

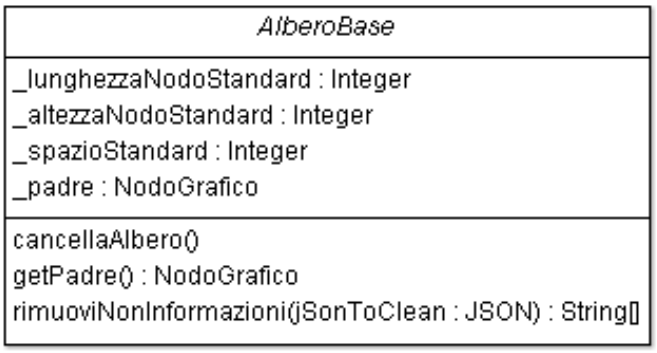

Figura 9: Diagramma di classe AlberoBase

4.2.7.1 Membri

- _lunghezzaNodoStandard: lunghezza che dovranno avere i vari nodi creati;
- _altezzaNodoStandard: altezza che dovranno avere i vari nodi creati;
- _spazioStandard: spazio minimo tra un nodo e un altro;
- _padre: riferimento alla radice dell'albero;

4.2.7.2 Funzionalità

- **cancellaAlbero()**: cancella l'albero disegnato;
- **rimuoviNonInformazioni(JSonToClean: JSON)**: ricava dal json un array di stringhe che contiene l'elenco delle chiavi che riguardano solo le informazioni e il nome, rimuovendo colore e figli.

4.2.8 AlberoMisto

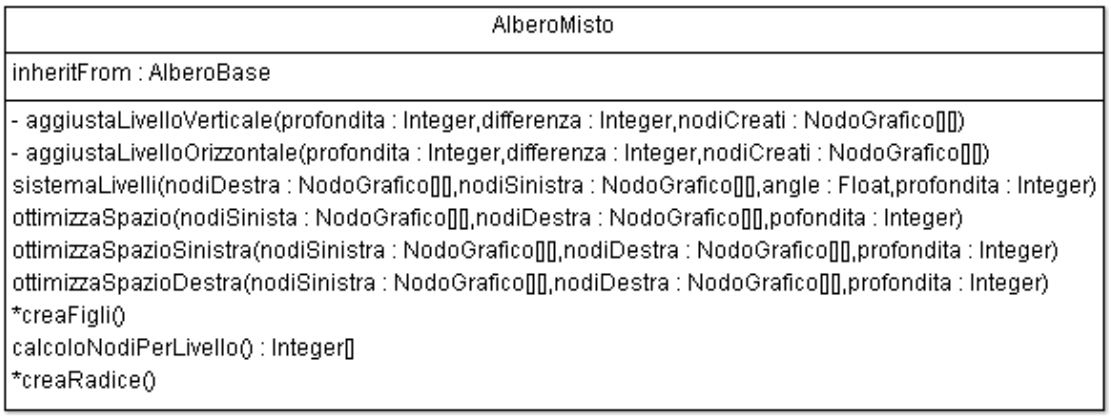

Figura 10: Diagramma di classe AlberoMisto

4.2.8.1 Membri

- inheritFrom: riferimento ad un oggetto di tipo AlberoBase. Visti i limiti del linguaggio javaScript l'ereditarietà è simulata invocando il costruttore di AlberoBase all'interno della definizione di AlberoMisto.

4.2.8.2 Funzionalità

- **aggiustaLivelloVerticale(profondita: Integer, differenza: Integer, nodiCreati: NodoGrafico[][])**: sposta un certo livello del sottoalbero sinistro pari a differenza (può essere un valore negativo), dopodiché si assicura che i livelli sottostanti siano almeno alla distanza minima dal livello modificato e richiama se stessa se non è così. *NodiCreati* è un array dove ogni elemento è l'array che contiene tutti i NodiGrafico di un certo livello;
- **aggiustaLivelloOrizzontale(profondita: Integer, differenza: Integer, nodiCreati: NodoGrafico[][])**: sposta un certo livello del sottoalbero destro pari a differenza (può essere un valore negativo), dopodiché si assicura che i livelli sottostanti siano almeno alla distanza minima dal livello modificato e richiama se stessa se non è così. *NodiCreati* è un array dove ogni elemento è l'array che contiene tutti i NodiGrafico di un certo livello;
- **sistemaLivelli(nodiDestra: NodoGrafico[], nodiSinistra: NodoGrafico[][], angle:float, profondita:Integer)**: funzione che si assicura che l'intero livello di un albero, in base al valore di *profondita*, sia alla distanza più grande tra lo spazio minimo tra un livello e l'altro e la coordinata affinché il livello non possa sforare nell'area dell'altro sottoalbero. *nodiDestra* e *nodiSinistra* sono array dove ogni elemento è l'array che contiene tutti i NodiGrafico di un certo livello di un certo sottoalbero;
- **ottimizzaSpazio(nodiSinistra: NodoGrafico[][], nodiDestra: NodoGrafico[][], profondita: Integer)**: Ottimizza lo spazio occupato dall'albero, permettendo ai livelli di sforare nell'area dell'altro sottoalbero se possibile, ovvero senza sovrapporsi ad altri nodi, la funzione considera dei due livelli quali dei due è più lontano dal livello precedente e cerca di spostare quello più distante. *nodiDestra* e *nodiSinistra* sono array dove ogni elemento è l'array che contiene tutti i NodiGrafico di un certo livello di un certo sottoalbero;

- **ottimizzaSpazioSinistra(nodiSinistra: NodoGrafico[][], nodiDestra: NodoGrafico[][], profondita: Integer)**: Ottimizza lo spazio occupato dal sottoalbero di sinistra, permettendo ai livelli di sforare nell'area dell'altro sottoalbero se possibile, ovvero senza sovrapporsi ad altri nodi, la funzione è invocata quando da una certa profondità in poi solo il sottoalbero sinistro esiste. *nodiDestra* e *nodiSinistra* sono array dove ogni elemento è l'array che contiene tutti i NodiGrafico di un certo livello di un certo sottoalbero;
- **ottimizzaSpazioDestra(nodiSinistra: NodoGrafico[][], nodiDestra: NodoGrafico[][], profondita: Integer)**: Ottimizza lo spazio occupato dal sottoalbero di destra, permettendo ai livelli di sforare nell'area dell'altro sottoalbero se possibile, ovvero senza sovrapporsi ad altri nodi, la funzione è invocata quando da una certa profondità in poi solo il sottoalbero destro esiste. *nodiDestra* e *nodiSinistra* sono array dove ogni elemento è l'array che contiene tutti i NodiGrafico di un certo livello di un certo sottoalbero;
- **creaFigli(nodiPerLivello: Integer[], profondita: Integer, figli: JSON, padre: NodoGrafico, canvas: RaphaelAdapter, offset: Integer, nodiCreati: NodoGrafico[][], tipo: String, dimSpostamento: Integer)**: crea i nodi per ogni livello di un certo sottoalbero, i parametri hanno il seguente significato:
  - nodiPerLivello, contiene i nodi che rimangono da disegnare per ogni livello;
  - profondita, la profondita a cui si è;
  - figli, i figli che rimangono da disegnare in formato JSON;
  - padre, il padre dei nodi che sono da disegnare;
  - canvas, riferimento da passare ai vari NodiGrafico affinché possano disegnare i vari oggetti;
  - offset, valore che stabilisce in base al parametro tipo quanto i nodi debbano essere disegnati distanti da un certo asse;
  - nodiCreati, i nodi creati fino all'invocazione del metodo creati;
  - tipo, il tipo, ovvero se il sottoalbero dev'essere disegnato come verticale o orizzontale;
  - dimSpostamento, ogni nodo che viene creato è spostato rispetto al precedente come minimo di questo valore.

  Restituisce offset calcolato in base ai nodi creati e alla loro posizione.
- **calcoloNodiPerLivello()** : restituisce un array dove ogni valore contiene il numero di nodi che saranno da creare ad una certa profondità;
- **creaRadice(Json: JSON, canvas: RaphaelAdapter, lunghezza: Integer, altezza: Integer, spazioTraNodi: Integer)** : crea il NodoGrafico radice e dopo richiama *creaFigli*.

4.2.9 AlberoVerticale

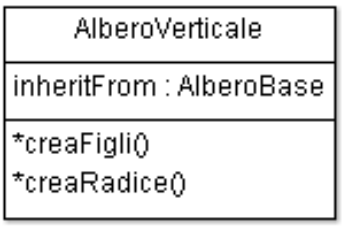

Figura 11: Diagramma di classe AlberoVerticale

4.2.9.1 Membri

- inheritFrom: riferimento ad un oggetto di tipo AlberoBase. Visti i limiti del linguaggio javaScript l'ereditarietà è simulata invocando il costruttore di AlberoBase all'interno della definizione di AlberoVerticale.

4.2.9.2 Funzionalità

- **creaFigli(nodiPerLivello: Integer[], profondita: Integer, figli: JSON, padre: NodoGrafico, canvas: RaphaelAdapter, offset: Integer, nodiCreati: NodoGrafico[][], tipo: String, dimSpostamento: Integer)**: crea i nodi per ogni livello di un certo sottoalbero, i parametri hanno il seguente significato:
  - profondita, la profondita a cui si è;
  - figli, i figli che rimangono da disegnare in formato JSON;
  - padre, il padre dei nodi che sono da disegnare;
  - canvas, riferimento da passare ai vari NodiGrafico affinché possano disegnare i vari oggetti;
  - offset, valore che stabilisce in base al parametro tipo quanto i nodi debbano essere disegnati distanti da un certo asse;
  - dimSpostamento, ogni nodo che viene creato è spostato rispetto al precedente come minimo di questo valore.

  Restituisce offset calcolato in base ai nodi creati e alla loro posizione.
- **creaRadice(Json: JSON, canvas: RaphaelAdapter, lunghezza: Integer, altezza: Integer, spazioTraNodi: Integer)** : crea il NodoGrafico radice e dopo richiama *creaFigli*.

4.2.10 AlberoClassico

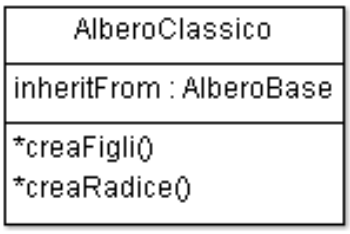

Figura 12: Diagramma di classe AlberoClassico

4.2.10.1 Membri

- inheritFrom: riferimento ad un oggetto di tipo AlberoBase. Visti i limiti del linguaggio javaScript l'ereditarietà è simulata invocando il costruttore di AlberoBase all'interno della definizione di AlberoClassico.

4.2.10.2 Funzionalità

- **creaFigli(nodiPerLivello: Integer[], profondita: Integer, figli: JSON, padre: NodoGrafico, canvas: RaphaelAdapter, offset: Integer, nodiCreati: NodoGrafico[][], tipo: String, dimSpostamento: Integer)**: crea i nodi per ogni livello di un certo sottoalbero, i parametri hanno il seguente significato:
  - profondita, la profondita a cui si è;
  - figli, i figli che rimangono da disegnare in formato JSON;
  - padre, il padre dei nodi che sono da disegnare;
  - canvas, riferimento da passare ai vari NodiGrafico affinché possano disegnare i vari oggetti;
  - offset, valore che stabilisce in base al parametro tipo quanto i nodi debbano essere disegnati distanti da un certo asse;
  - nodiCreati, i nodi creati fino all'invocazione del metodo creati;
  - dimSpostamento, ogni nodo che viene creato è spostato rispetto al precedente come minimo di questo valore.

  Restituisce offset calcolato in base ai nodi creati e alla loro posizione.
- **creaRadice(Json: JSON, canvas: RaphaelAdapter, lunghezza: Integer, altezza: Integer, spazioTraNodi: Integer)** : crea il NodoGrafico radice e dopo richiama *creaFigli*.

4.3 Algoritmo di creazione Albero

Il diagramma di attività in figura 13 a pagina 22 descrive come verrà disegnato un albero generico partendo dall'oggetto JSON ottenuto.

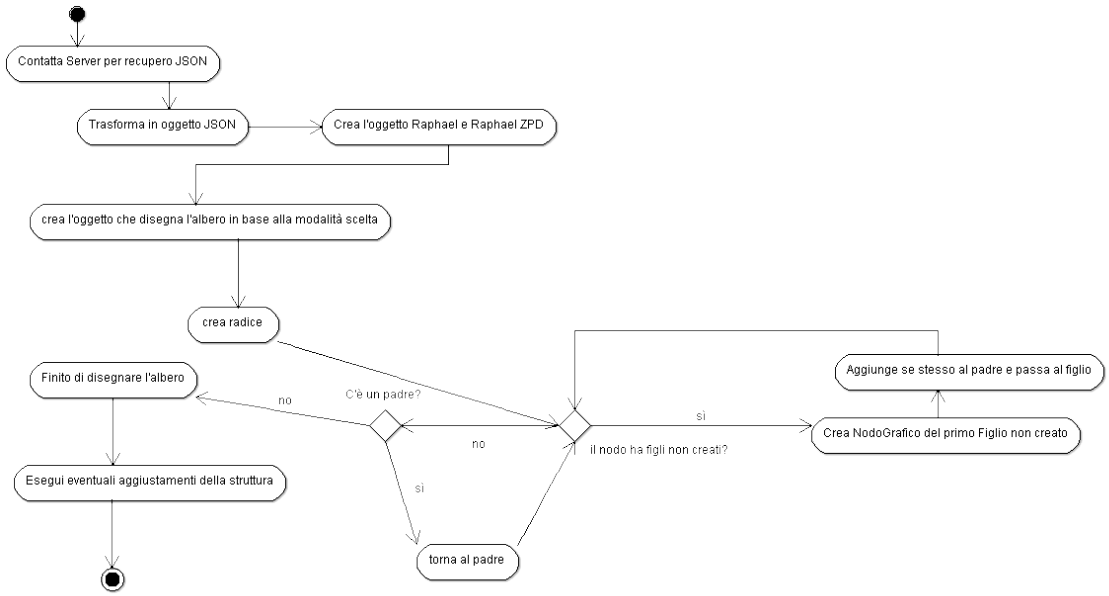

Figura 13: Diagramma di attività Algoritmo creazione albero

L'algoritmo è semplice, viene contattato il server per il recupero di un file formattato secondo le direttive JSON tramite l'invocazione della funzionalità *recuperoJSON*, viene quindi trasformato

in un oggetto JSON, creata la radice e aggiunta alla tabella delle statistiche le informazioni sul nodo. Successivamente viene invocato il metodo *creaNodo* che si occuperà di creare il nodo stesso, associarlo al padre e se il nodo stesso ha dei figli chiamerà ricorsivamente la funzione stessa sul primo figlio. Se un nodo non ha figli, ritorna al padre, che disegna il figlio successivo (se presente) oppure ritorna al livello successivo dell'albero. L'algoritmo termina quando si è tornati alla radice e i figli della radice sono tutti disegnati.

5 Design Pattern

5.1 Adapter

Durante la fase di progettazione è stato deciso di utilizzare il pattern strutturale *Adapter*. L'uso di questo pattern diventa utile quando interfacce di classi differenti devono poter comunicare tra loro. La struttura rappresentata in figura struttura del pattern *Adapter* rappresenta un *Object Adapter* ovvero utilizza la composizione tra oggetti e non l'ereditarietà. I Partecipanti del pattern *Adapter* sono:

Adaptee : definisce l'interfaccia che ha bisogno di essere adattata;

Target : definisce l'interfaccia che usa il Client;

Client : collabora con gli oggetti in conformità con l'interfaccia *Target*;

Adapter : adatta l'interfaccia *Adaptee* all'interfaccia *Target*.

Nel contesto di questo progetto il pattern *Adapter* viene usato due volte e serve a rendere l'interfaccia indipendente dalle librerie utilizzate, ovvero Raphael e RaphaelZPD. In questo modo sarà possibile cambiare la libreria grafica senza dover modificare l'interfaccia che utilizza il *Client*.