

“Software Engineering”

Course

a.a. 2016-2017

Template version 1.0

Lecturer: Prof. Henry Muccini (henry.muccini@univaq.it)

Planner Path Calculator version v3

Deliverables

Date	<20/01/2017>
Deliverable	V3
Team (Name)	Valentini+

Team Members		
Name & Surname	Matriculation Number	E-mail address
Valentini Stefano	227718	stefano.valentini2@student.univaq.it
Battista Federico	237121	federico.battista@student.univaq.it
Cecchini Valentina	227719	valentina.cecchini@student.univaq.it
Di Gregorio Zitella Luca	229334	luca.digregoriozitella@student.univaq.it
Perelli Andrea	236059	andrea.perelli@student.univaq.it

Project Guidelines

[do not remove this page]

This page provides the Guidelines to be followed when preparing the report for the Software Engineering course. You have to submit the following information:

- *This Report*
- *Diagrams (Analysis Model, Component Diagrams, Sequence Diagrams, Entity Relationships Diagrams)*
- *Effort Recording (Excel file)*

Important:

- *document risky/difficult/complex/highly discussed requirements*
- *document decisions taken by the team*
- *iterate: do not spend more than 1-2 full days for each iteration*
- *prioritize requirements, scenarios, users, etc. etc.*

Project Rules and Evaluation Criteria

General information:

- *This homework will cover the 80% of your final grade (20% will come from the oral examination).*
- *The complete and final version of this document shall be not longer than 40 pages (excluding this page and the Appendix).*
- *Groups composed of seven students (preferably).*

I expect the groups to submit their work through GitHub

Use the same file to document the various deliverable.

Document in this file how Deliverable “i+1” improves over Deliverable “i”.

Project evaluation:

Evaluation is not based on “quantity” but on “quality” where quality means:

- *Completeness of delivered Diagrams*
- *(Semantic and syntactic) Correctness of the delivered Diagrams*
- *Quality of the design decisions taken*
- *Quality of the produced code*

Table of Contents of this deliverable

List of Challenging/Risky Requirements or Tasks	4
A. Requirements Collection	6
<i>Use Case Diagram</i>	<i>7</i>
<i>A.2 Non Functional Requirements</i>	<i>12</i>
<i>A.3 Content</i>	<i>12</i>
<i>A.4 Assumptions</i>	<i>13</i>
<i>A.5 Prioritization</i>	<i>13</i>
B. Analysis Model	15
<i>B.1 Boundary Object.....</i>	<i>15</i>
<i>B.2 Entity Object</i>	<i>16</i>
<i>B.3 Controller Object</i>	<i>16</i>
C. Software Architecture.....	18
<i>C.1 The static view of the system: Component Diagram.....</i>	<i>19</i>
<i>C.2 The dynamic view of the software architecture: Sequence Diagram</i>	<i>21</i>
<i>Sequence Diagram of Abstract View of the System.....</i>	<i>22</i>
<i>Sequence Diagram Create.....</i>	<i>24</i>
<i>Sequence Diagram Delete</i>	<i>26</i>
<i>Sequence Diagram Sum.....</i>	<i>28</i>
<i>Sequence getTrees</i>	<i>29</i>
<i>Sequence Diagram GraphAware.....</i>	<i>30</i>
D. ER Design.....	31
E. Class Diagram of the implemented System.....	33
F. Design Decisions.....	36
G. Explain how the FRs and the NFRs are satisfied by design	41
H. Effort Recording	43
Appendix. Code.....	45

- Applicativo **disponibile all'indirizzo**: <http://94.177.184.175/index.php>
- **Video dimostrativo** del prototipo: <https://www.youtube.com/watch?v=OSWNCuitlco>
(nella descrizione del video è presente la timeline degli eventi con una breve spiegazione)

List of Challenging/Risky Requirements or Tasks

Challenging Task	Date the task is identified	Date the challenge is resolved	Explanation on how the challenge has been managed
Scelta di un DB adatto allo scopo del Sistema	01/11/2016	04/11/2016	Al primo incontro ci siamo soffermati sulla scelta di un Database adatto al nostro scopo: da una parte c'era MSSQL che il committente ci ha suggerito, non conosciuto dai membri del gruppo, dall'altra alcuni hanno proposto di pensare all'impiego di un DB NoSQL a Grafo (anche questo sconosciuto) data la particolare affinità tra questo progetto e la strutturazione dei dati nei DB a Grafo. Alla fine la nostra scelta è ricaduta sul Database NoSQL a Grafo Neo4j.
Scelta del database NoSQL Neo4j che nessuno del team conosce	04/11/2016	21/11/2016	Nella suddetta data, siamo arrivati alla conclusione che il DB scelto può effettivamente darci le performance richieste, dopo aver fatto learning ed alcuni primitivi test. Rimane, tuttavia, l'incognita della concorrenza che, non avendo le risorse adeguate per i test, non può essere attualmente risolta.
Performance – Concorrenza/ Accessi Multipli	01/11/2016	19/12/2016	È stato risolto anche questo aspetto in suddetta data. Il prototipo che avevamo a disposizione, testato solo su una singola macchina in locale (poiché risultava difficile poter effettuare test su un numero rilevante di Concorrenze), si è rilevato avere qualche problema durante la creazione dell'albero, non tanto nella creazione dei nodi ma nella creazione degli Index (vedi Design Decision). Infatti, per la suddetta il DB utilizza i Lock su Read/Write e, per un certo numero di concorrenze, ricadeva in un Deadlock. La soluzione a cui si è arrivati consiste in un Processo Server che rimane in ascolto per i Client (i processi degli utenti che richiedono la creazione degli alberi tramite la GUI) ed una alla volta, accodando tutte le richieste in arrivo, genera gli Index sui Vertici in creazione, evitando il Deadlock.
Modifica dell'albero	01/11/2016	05/11/2016	Fin da subito non è stato molto chiaro se il Sistema da progettare avesse dovuto prevedere la modifica di un albero creato in precedenza. Ciò era un punto cruciale per le scelte che si sarebbero dovute fare. Con il committente, in seguito, è stato pattuito che la modifica non fosse una funzionalità necessaria del Sistema.

A. Requirements Collection

A.1 Functional Requirements

<List and describe functional requirements through Use Case Diagrams. Then, prioritize them, and provide a table-based description of the most important requirements>

A1.1 GUI Requirements

A1.1.a – Il Sistema dovrà prevedere una GUI web-oriented (HTML5).

A1.1.b – Sarà la GUI a permettere l'interazione tra l'utente e le funzionalità del sistema (creazione, eliminazione di alberi, somma ecc.).

A1.1.c – La GUI dovrà permettere l'inserimento dei seguenti parametri, necessari per la creazione dell'albero:

- **Split Size:** numero esatto di figli che ogni nodo (eccetto le foglie) avrà;
- **Depth:** l'altezza dell'albero;
- **AttributeList:** lista di attributi associati a Vertex ed Edge:
 - VertexAttributeList;
 - EdgeAttributeList.
- **Random(K, N) :** indicherà all'engine in che range (tra K e N, con K e N numeri Reali) generare il valore random di un attributo. Andrà specificato per ogni attributo precedentemente definito;
- **Nome dell'albero:** nome con cui l'albero verrà salvato.

A1.1.d – Permetterà la definizione di un insieme arbitrariamente grande di attributi sia su Vertex che su Edge.

A1.1.e – Permetterà di “visualizzare” (Vedi Design Decision) un albero precedentemente creato presente sul Database.

A1.1.f – Attraverso la GUI sarà possibile richiedere l'operazione di Somma sull'Albero selezionato, scegliendo due Vertici ad esso appartenenti.

A1.1.g – La GUI dovrà mostrare l'output dell'operazione di Somma sull'Albero insieme al tempo impiegato per calcolarlo.

A1.2 Business Logic Requirements

A1.2.a – Il Sistema dovrà essere in grado di modellare grafi ad albero.

A1.2.b – Ogni grafo verrà rappresentato in termini di Vertex (Nodi) ed Edge (Archi).

A1.2.c – Il Sistema dovrà permettere la “gestione” degli alberi:

A1.2.c.1 – Creazione: vincolata all’inserimento dei parametri descritti in A1.1.c;

A1.2.c.2 – Eliminazione di un albero selezionato;

A1.2.c.3 – Visualizzazione/Selezione: deve essere in grado di recuperare dall’albero le informazioni principali (Vedi Design Decision).

A1.2.d – Il Sistema prevederà un metodo per generare in maniera random i valori degli attributi di nodi ed archi, nei range inseriti dall’utente.

A1.2.e – Il Sistema dovrà prevedere un’Operazione di Somma sull’Albero così definita: dati due Vertici A e B di un albero, scelti dall’utente, il sistema dovrà ritornare la lista dei vertici attraversati dal cammino da A a B insieme alla somma per ogni attributo su tutti i vertici e archi del cammino.

A1.3 DB Requirements

A1.3.a – Tutti gli alberi creati andranno salvati sul database.

A1.3.b – Il DB dovrà permettere di accedere ad un albero creato in precedenza.

A1.3.c – Dovrà essere strutturato in maniera tale da permettere il corretto salvataggio e il corretto funzionamento delle operazioni di somma.

Use Case Diagram

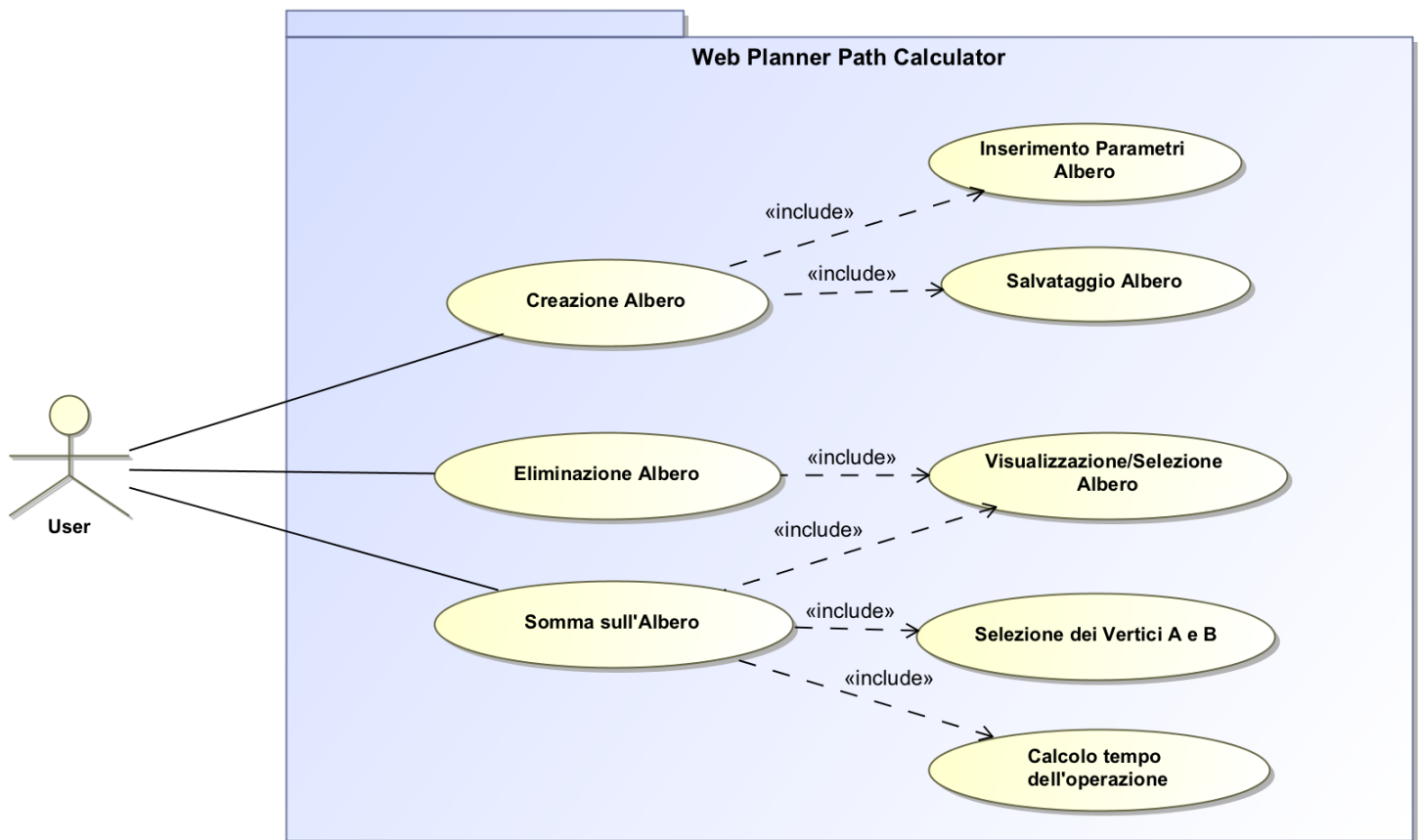


Figura A.1: Use Case Diagram che rappresenta le funzionalità principali del sistema.

USE CASE	REQUISITI FUNZIONALI COPERTI
Creazione Albero	A1.1.b, A1.2.c.1
Eliminazione Albero	A1.1.b, A1.2.c.2
Somma sull'Albero	A1.1.b, A1.1.f, A1.1.g, A1.2.e
Inserimento parametri Albero	A1.1.c, A1.1.d, A1.2.d
Salvataggio Albero	A1.3.a, A1.3.c
Visualizzazione/Selezione Albero	A.1.1.e, A1.1.f, A1.2.c.3, A1.2.e, A1.3.b
Selezione dei Vertici A e B	A1.1.f, A1.2.e
Calcolo tempo dell'operazione	A1.1.g

USE CASE #	Creazione Albero	
Goal in Context	Modulo che permette la Creazione di un albero	
Scope & Level	Primary Task, è il modulo che si interfaccia con l’user	
Preconditions	Per una corretta creazione dell’albero, è necessario prima l’esecuzione del modulo “Inserimento parametri albero”.	
Success End Condition	Se la creazione va a buon fine, l’albero verrà salvato sul database, attraverso il modulo “Salvataggio Albero”.	
Failed End Condition	Se la creazione dovesse fallire, lo stato dell’intero sistema non verrà affetto da alcun cambiamento.	
Primary Actors	User	
Trigger	Richiesta da parte dell’utente	
DESCRIPTION	Step	Action
	1	L’User si interfaccia col modulo creazione
	2	Viene chiamato il modulo Inserisci Parametri Albero, permettendo allo User di istanziare l’albero
	3	L’albero viene creato e salvato sul Database.

Priority:	E’ l’operazione più onerosa dal punto di vista temporale.
Performance	Il tempo massimo accettabile per la creazione di un albero è di 1 ora
Frequency	1 volta a settimana per utente

USE CASE #	Somma Sull'Albero	
Goal in Context	Dati un nodo di partenza, uno di arrivo e un elenco di attributi su nodi/archi si effettua la somma sugli attributi sul percorso dal nodo di partenza a quello di arrivo.	
Scope & Level	Primary Task, è accessibile dall'user	
Preconditions	Deve esistere l'albero che verrà selezionato su cui si intende far la somma e deve esistere un cammino dal nodo di partenza e il nodo di arrivo.	
Success End Condition	Lo stato del sistema rimane invariato, vengono restituiti i valori delle somme, il tempo di computazione e la lista di Vertici appartenenti al cammino.	
Failed End Condition	Lo stato del sistema rimane invariato	
Primary Actor	User	
Trigger	Richiesta da parte dell'utente.	
DESCRIPTION	Step	Action
	1	L'utente si interfaccia con il modulo di Somma.
	2	L'utente seleziona un albero tra quelli presenti nel Sistema.
	3	Vengono inseriti il nodo di partenza e quello di destinazione.
	4	Vengono calcolate le somme e la lista di nodi del cammino.
	5	Attraverso il modulo Calcolo Tempo dell'Operazione verrà mostrato come risultato anche il tempo impiegato per la computazione della somma.

Priority:	E' l'operazione più critica del sistema, quella per cui ci sono stati imposti più vincoli.
Performance	Sono richiesti non più di 30 secondi per operazioni di somma su alberi da 1.000.000 di nodi; non più di 60 secondi per alberi da 2.000.000 di nodi
Frequency	3 volte al giorno per utente

USE CASE #	Eliminazione Albero	
Goal in Context	Modulo che permette l'eliminazione di un albero	
Scope & Level	Primary Task, è il modulo che si interfaccia con l'user	
Preconditions	Per una corretta eliminazione dell'albero, è necessario prima l'esecuzione del modulo "Selezione/Visualizzazione Albero".	
Success End Condition	Se la creazione va a buon fine, l'albero verrà salvato sul database, attraverso il modulo "Salvataggio Albero".	
Failed End Condition	Se l'eliminazione dovesse fallire, lo stato dell'intero sistema non sarà affetto da alcun cambiamento.	
Primary Actors	User	
Trigger	Richiesta da parte dell'utente	
DESCRIPTION	Step	Action
	1	L'User si interfaccia col modulo Eliminazione.
	2	L'User seleziona l'albero da eliminare.
	3	L'albero viene eliminato.

A.2 Non Functional Requirements

A2.1 - Efficiency NFR:

A2.1.a - Performance:

- Il Sistema dovrà essere “veloce” nella risposta alle operazioni, disambighiamo il concetto di veloce:
- La creazione dell'albero verrà effettuata, in media, una volta alla settimana, quindi è ragionevole un tempo di attesa più lungo;
- Per quanto riguarda la somma, essa viene ritenuta più critica ed è la più frequente (3 volte al giorno circa), quindi si presterà maggiore attenzione alla sua performance; il committente ha richiesto che la somma dovrà rispondere in circa 30 secondi per alberi da 1.000.000 di nodi, 60 secondi per alberi da 2.000.000.

Oltre queste grandezze non ci si aspetta una scalabilità lineare.

- Il Sistema dovrà essere in grado di gestire un numero di accessi concorrentiali che vari tra 10 e 100.

A2.1.b - Space:

- Il committente non ha posto alcun vincolo sullo spazio che dovrà al più occupare il nostro sistema, ma ci ha anche consigliato di occupare quanto più spazio ci serve per rendere veloce le operazioni di selezione e di somma.

A2.2 - Security:

- Il committente sostiene che l'ambiente dove verrà utilizzato il nostro software è un ambiente sicuro, quindi non sarà nostro compito particolare preoccuparci di ciò.

A2.3 - Usability:

- La GUI deve essere estremamente user-friendly, l'utente dovrà avere “a portata di mano” tutte le funzionalità del sistema, soprattutto quelle più frequenti (selezione e somma per esempio).

A2.4 - Tecnology Requirements:

- La GUI dovrà essere implementata come una Web Interface basata su HTML5.
- Il committente ci ha proposto di adottare MSSQL come Database Technology. Altri DBMS possono essere scelti a patto che siano Open-source.

A.3 Content

*<Describe the **data provenance** (use of external API, web service, DB ...)>*

Utilizzeremo un Database NoSQL a Grafo, Neo4j poiché abbiamo valutato che può fornirci delle performance di tempo, maggiori rispetto ad un DB SQL. Inoltre l'implementazione dei grafi ci viene fornita "gratuitamente" dal DB stesso. (Vedi Design Decision)

Inoltre, per la comunicazione tra Engine e Neo4j utilizzeremo delle API per PHP messe a disposizione open-source da GraphAware.

A.4 Assumptions

<Briefly document, in this section, the most relevant requirement assumptions/decisions you had to make during your project>

1. Assunzioni per i vertici

A4.1.a – Ai vertici verrà assegnato un nome Vertex_k con k numero progressivo, Vertex_1 è sempre la radice.

2. Assunzioni per gli attributi

A4.2.a – Gli attributi essendo numeri Reali si è scelto di assegnargli una precisione di 3 cifre dopo la virgola.

3. Assunzione per l'albero

A4.3.a – Gli archi sono sempre orientati dai figli verso i padri.

A.5 Prioritization

Requisiti listati secondo ordine di priorità.

A5.1 - **Creazione dell'albero:**

Il sistema nasce per la gestione di grafi ad albero, quindi si è ritenuto di assegnare priorità "massima" ad esso; inoltre la creazione dell'albero è cruciale per la richiesta della somma.

A5.2 - **Calcolo della Somma sull'albero:**

L'operazione principale che verrà effettuata sugli alberi generati. Da una prima analisi essa sarà un'operazione critica per quanto riguarda le performance (velocità e concorrenza) pertanto si è deciso di assegnare anche ad essa priorità alta.

A5.3 - **Eliminazione dell'albero**

È l'operazione con priorità più bassa, in quanto non è fondamentale per il funzionamento del sistema.

B. Analysis Model

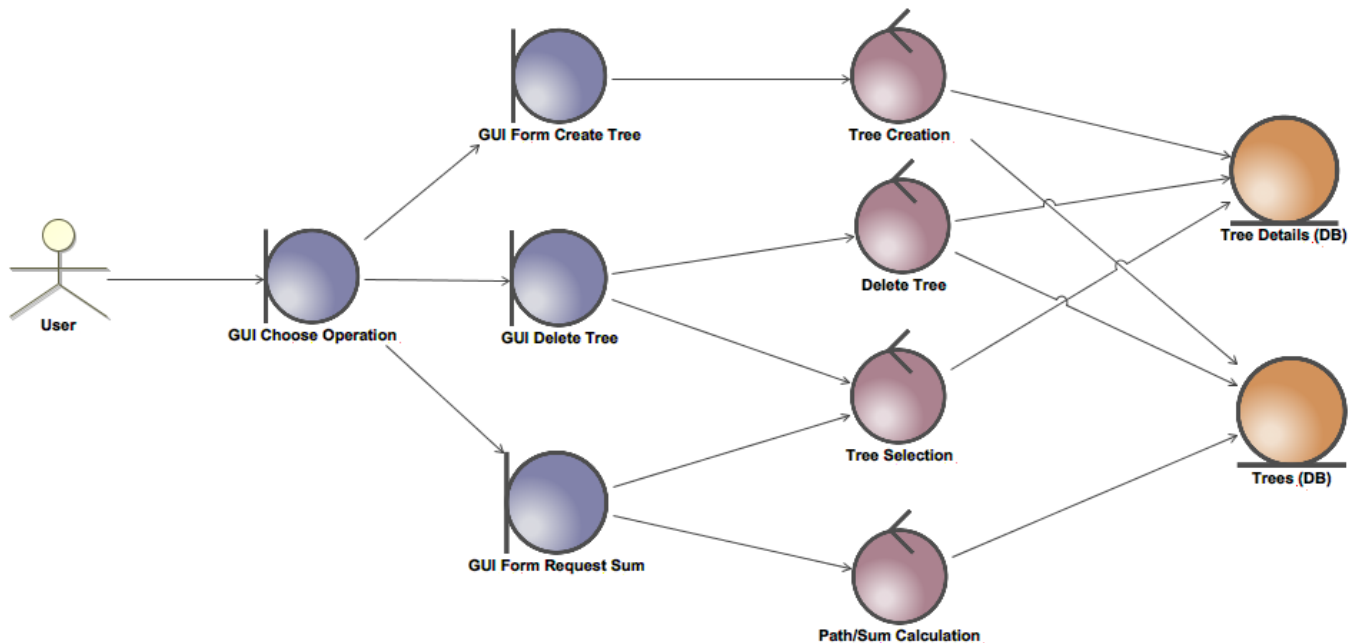


Figura B.1 Analysis Model del Sistema

B.1 Boundary Object

GUI Choose Operation

Sarà la GUI alla quale l'Utente avrà diretto accesso per la scelta dell'operazione che esso vorrà effettuare. GUI Choose Operation darà accesso alle seguenti "sotto-interfacce": GUI Create, GUI Delete, GUI Sum.

GUI Form Create

È l'interfaccia che richiederà la creazione dell'albero, permettendo all'Utente di inserire gli attributi necessari alla creazione che verranno poi utilizzati dal Control "Tree Creation" per la corretta creazione.

GUI Delete

È l'interfaccia che permette di selezionare, attraverso il Control Object Tree Selection, un albero tra quelli già presenti sul Database e richiederne l'eliminazione che spetterà al Control Object Tree Delete.

GUI Form Request Sum

È l'interfaccia che permette di selezionare un albero precedentemente creato attraverso il Control "Tree Selection", risiedente nel DB, e richiederne la somma sugli attributi da un vertice di partenza ad un vertice destinazione scelti dall'Utente.

B.2 Entity Object**Trees (DB)**

Gli Entity Object nel nostro Sistema saranno gli alberi creati e salvati sul Database. Essi dovranno essere disponibili all' "Engine" che si occuperà dell'operazione di somma.

Tree Details (DB)

Conterrà le informazioni utili di ogni albero, esse serviranno al Controller Tree Selection per semplificare l'operazione di Selezione/Visualizzazione. Questi Entity verranno tenuti aggiornati dai Controller Tree Creation e Tree Deletion.

B.3 Controller Object**Tree Creation**

Controller relativo alla creazione dell'albero, sarà il "motore" che comunicando con l'interfaccia relativa farà sì che l'albero di cui si è richiesta la creazione risiederà sul Database.

Tree Selection

Controller che, ogniqualvolta sarà richiesto, recupererà tutte le informazioni principali di tutti gli alberi effettivamente presenti sul database e, comunicando con la relativa GUI, permetterà all'utente di sceglierne uno per l'operazione richiesta.

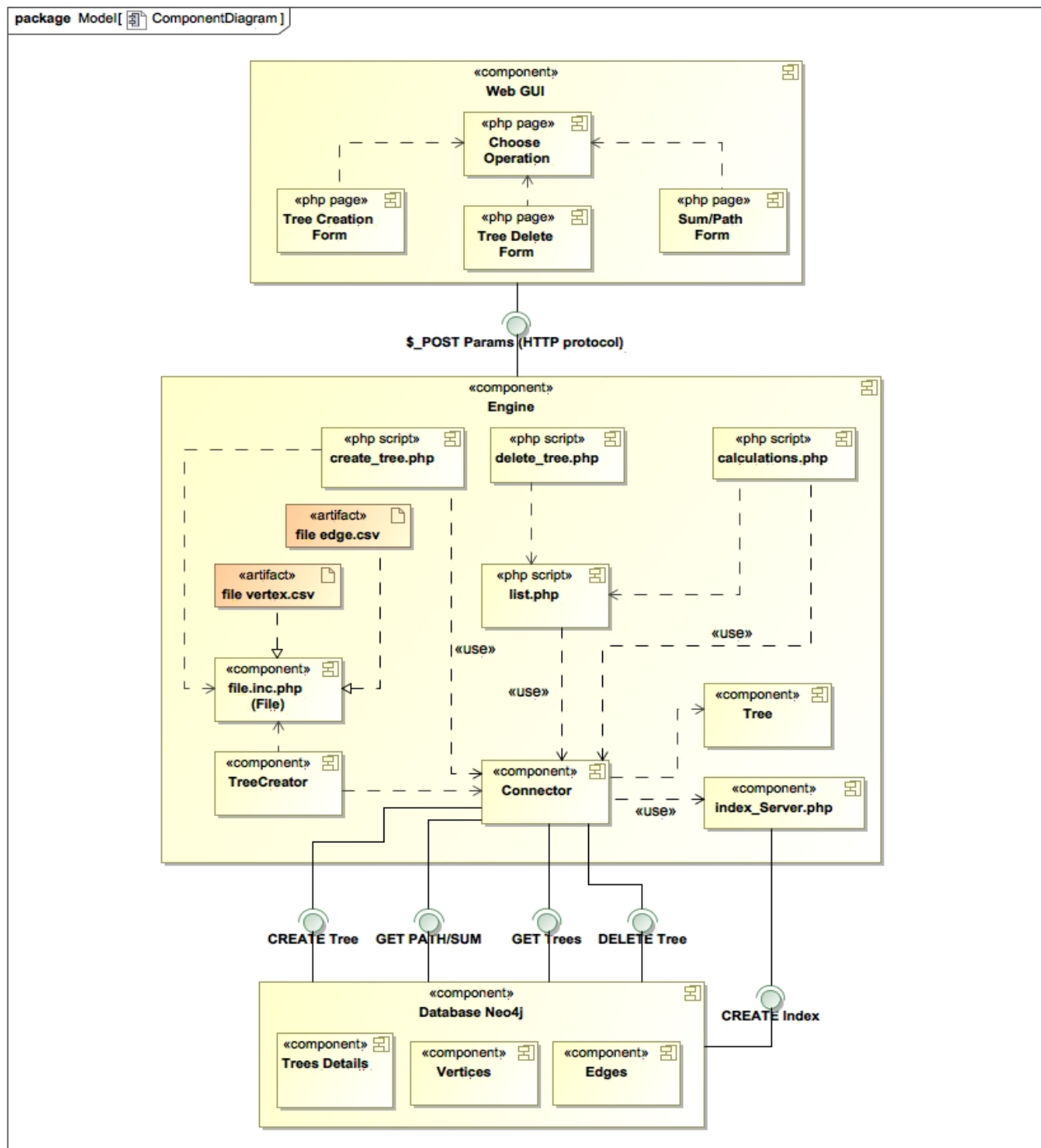
Delete Tree

Modulo dedicato alla richiesta di cancellazione: farà sì che l'albero designato per la cancellazione non sarà più disponibile per le altre operazioni. Delegherà l'operazione di cancellazione al Database che provvederà a cancellarlo definitivamente dal Sistema.

Path/Sum Calculation

Controller che gestirà l'operazione fulcro del sistema: passatogli l'albero, il vertice di partenza e quello di destinazione attraverso la relativa GUI, questo modulo dovrà essere in grado di ritornare all'utente il percorso dal vertice di partenza al vertice di destinazione (se esiste), insieme alla somma su ogni attributo di ogni vertice appartenente al percorso. Questo "processo" sarà inoltre delegato al calcolo del tempo impiegato per tale operazione.

C. Software Architecture



C.1 The static view of the system: Component Diagram

Il Sistema è composto da 3 macro componenti:

- **Web GUI:** formata da una pagina principale nella quale può essere scelta l'operazione da effettuare, e altre 3 pagine rispettivamente per inserire i dati relativi alla creazione di un albero, per scegliere un albero da cancellare e per scegliere un albero e i nodi tra cui calcolare il cammino e le somme degli attributi.
- **Engine:** modulo dedicato allo svolgimento delle operazioni.
 - L'engine nel caso della creazione, delega "create_tree.php"; il quale produce due file ".csv" (tramite "file.inc.php", uno per i nodi ed uno per gli archi) che poi verranno usati dal database per istanziare l'albero. Sarà il component TreeCreator a richiedere la creazione dell'albero, comunicando con il DB tramite Connector e la relativa interfaccia CREATE Tree.

Per l'interazione tra Connector e il component "index_server.php" si vedano le Design Decision.
 - L'engine nel caso della somma, delega "calculations.php"; il quale effettuerà due passaggi:
 - tramite "list.php" verrà recuperata la lista di tutti gli alberi presenti nel DB. Ciò è possibile grazie all'interazione di Connector con l'interfaccia GET Trees.
 - la somma verrà effettuata, sull'albero selezionato, da Connector tramite l'interfaccia GET PATH/SUM.

In questo scenario non c'è interazione tra "Connector" e il component "index_server.php".
 - L'engine nel caso della cancellazione, delega "delete_tree.php"; il quale effettuerà due passaggi:
 - come per la somma, sarà "list.php" a recuperare la lista di tutti gli alberi presenti nel DB.
 - la cancellazione verrà effettuata, sull'albero selezionato, da Connector tramite l'interfaccia DELETE Tree.

Per l'interazione tra Connector e il component "index_server.php" si vedano le Design Decision.

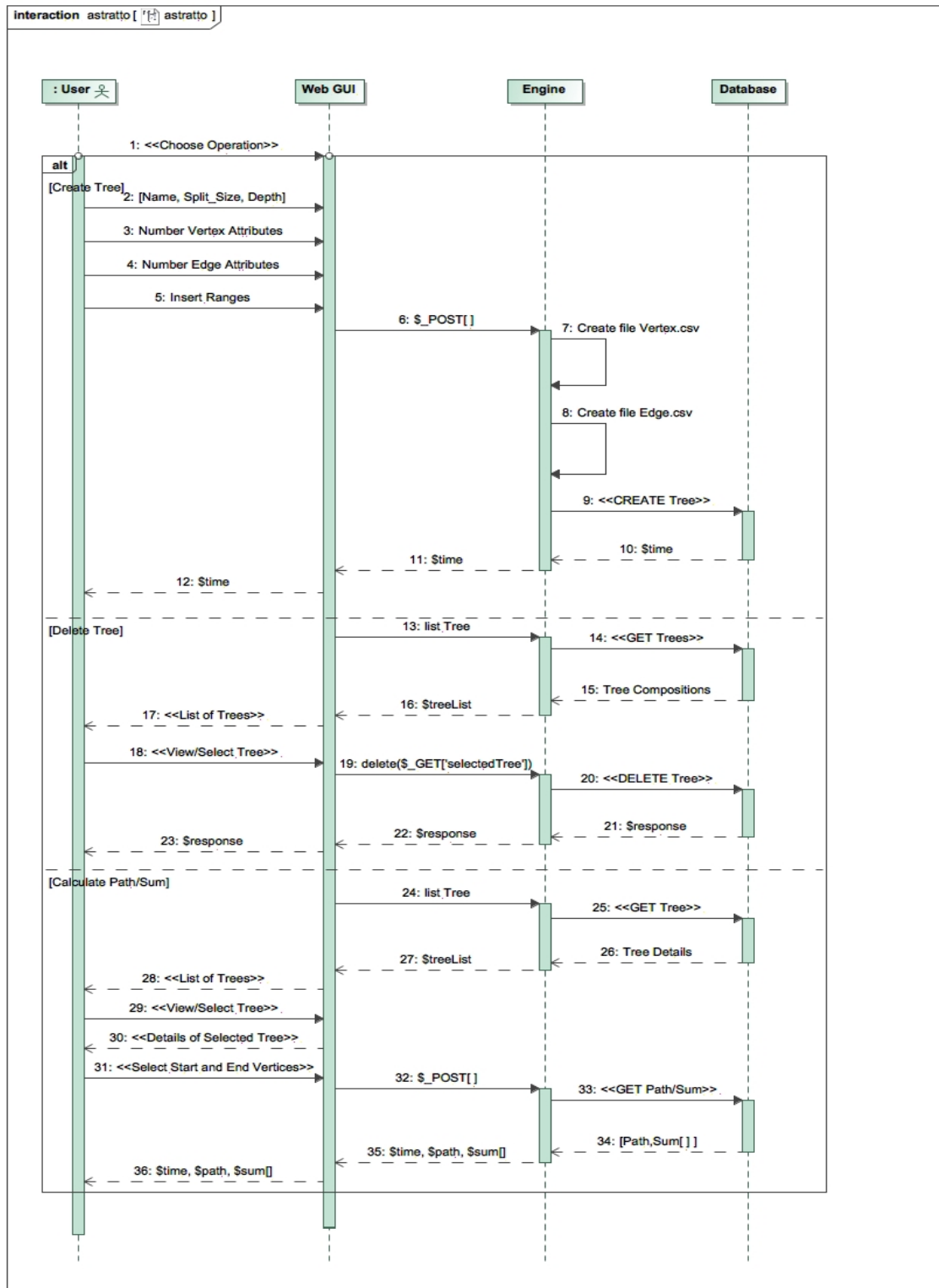
- **Database NoSql Neo4j** in cui verranno salvati tre tipi di dato:
 - Tree Details: Per ogni albero verranno salvati i loro dettagli principali necessari per l'interfaccia GET Trees.
 - Vertex: rappresenteranno tutti i nodi divisi per ogni albero, ognuno di esso con i relativi attributi.
 - Edge: rappresenteranno tutti gli archi anche loro divisi per ogni albero, ognuno con i relativi attributi.

Per maggiori informazioni riguardo l'effettivo salvataggio dei dati sul database, si veda la sezione D di questo Deliverable.

Le varie sottopagine della Web GUI sono raggiungibili solo dalla pagina principale, le sottopagine richiedono le relative interfacce, tramite le quali si potrà usufruire dei servizi dell'engine dedicati alla specifica operazione richiesta.

L'engine costruisce le query da somministrare al database tramite il linguaggio di query di Neo4j.

C.2 The dynamic view of the software architecture: Sequence Diagram



Sequence Diagram of Abstract View of the System

Il seguente sequence mostra la sequenza di operazioni effettuate dal sistema di fronte alla richiesta del cliente finale. Precisiamo che questo primo sequence diagram si trova all'interno di un alternative combined fragment, questo perché all'utente verrà richiesto di scegliere una delle possibili operazioni (presenti nella GUI) di: Creazione, Cancellazione, Somma.

Di seguito vengono esplicitati i seguenti casi:

- Creazione Albero: All'utente verrà richiesto di inserire Name, Split_Size, Depth, Number Vertex Attribute, Number Edge Attribute e i Ranges. Una volta effettuata questa operazione i dati vengono inviati con il metodo **\$_POST** all'engine, che fornisce le query necessarie al completamento dell'operazione. Quest'ultima si divide in due fasi:
In una prima vengono creati i due file (vertex_NomeAlbero e edge_NomeAlbero).csv dove verranno memorizzate le informazioni relative ad ogni singolo albero.
Nella seconda fase viene eseguita la query che crea fisicamente la struttura dell'albero, prendendo in input i file .csv creati precedentemente. Infine verrà restituito all'utente il successo o meno dell'operazione con il relativo tempo totale di esecuzione.
- Cancellazione Albero: L'utente seleziona dalla GUI l'operazione di cancellazione, che provvederà a richiedere all'engine di lanciare la query GET_trees, la quale restituirà la lista degli alberi presenti nel database, che verrà riportata alla corrispondente Web GUI per mostrarla all'utente.
A questo punto l'utente potrà scegliere l'albero da eliminare, il flusso di controllo passa all'engine il quale genera e somministra la query di cancellazione al database, tornando come risultato l'esito ed il tempo impiegato.
- Somma Albero: L'utente seleziona dalla GUI l'operazione di somma; quest'ultima richiederà la lista degli alberi all'engine, il quale provvederà ad interagire con il Database e tramite l'interfaccia GET Trees otterrà la lista degli alberi presenti nel sistema con i relativi dettagli, che passerà alla GUI e verrà messa a video.


A questo punto l'utente può selezionare l'albero su cui effettuare la somma.

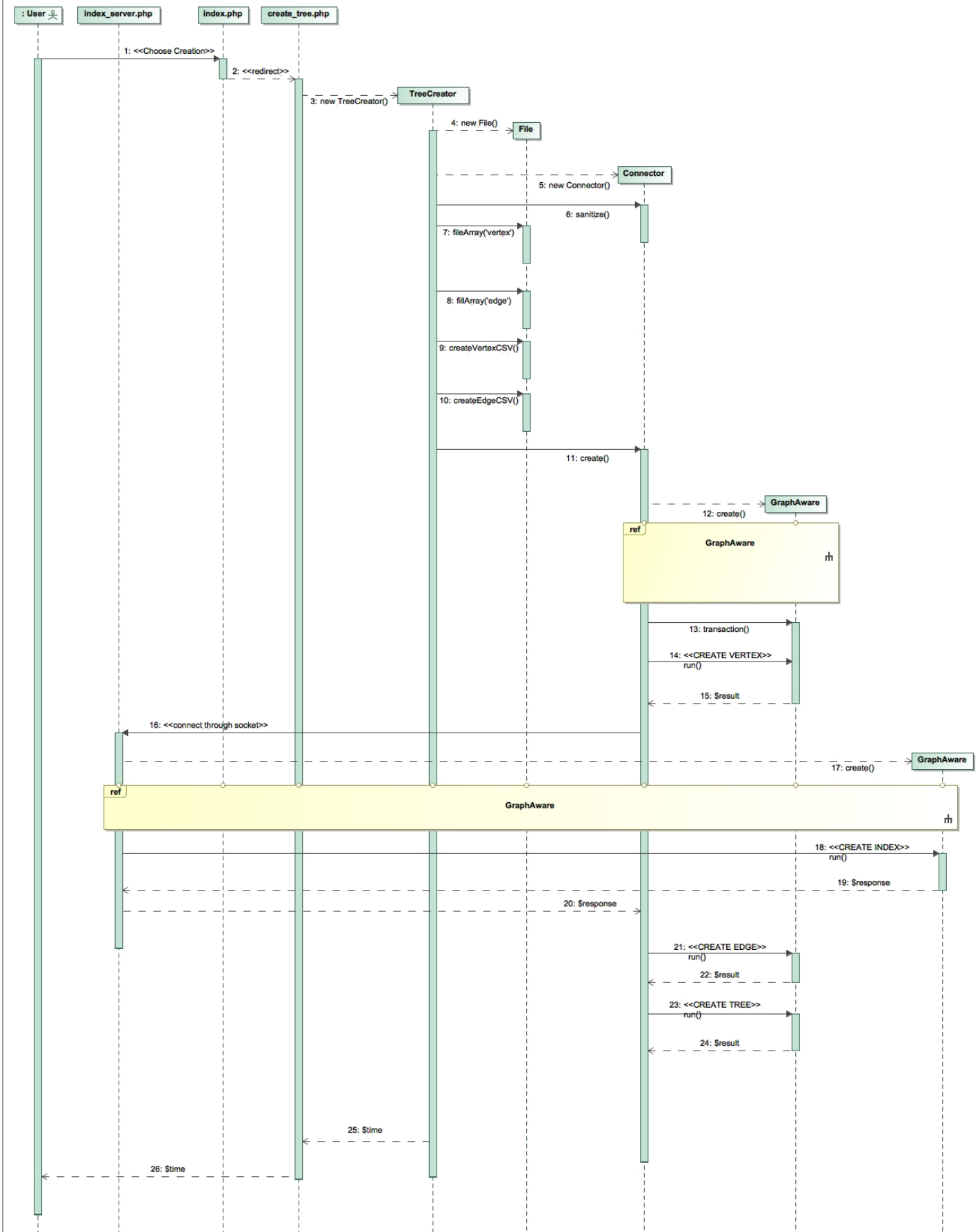
La selezione fa sì che l'utente, venga reindirizzato alla pagina web, in cui potrà scegliere i due vertici che segnano l'inizio e la fine del path su cui effettuare il calcolo.

In seguito i dati inseriti dall'utente verranno inviati tramite **\$_POST** all'engine, il quale li utilizzerà per la costruzione della query che verrà trasmessa al database.

Quest'ultimo risponderà all'engine tornando i risultati delle somme e i nodi con i relativi attributi, appartenenti al path scelto.

L'engine restituirà tali risultati ed il tempo impiegato alla web GUI, affinché vengano visualizzati dall'utente.

Interaction create [ create]



Sequence Diagram Create

L'utente dopo aver scelto "Create" dalla pagina index.php (oppure nel menu laterale) viene reindirizzato alla pagina create_tree.php che permetterà l'inserimento dei dati relativi all'albero.

La pagina create_tree.php istanzia un nuovo oggetto TreeCreator il quale istanzia un nuovo oggetto File, quest'ultimo istanzia un oggetto Connector.

TreeCreator chiama il metodo sanitize() di Connector che ha il compito di leggere i dati inviati dalla form e di ripulirli da caratteri speciali che potrebbero far fallire la query.

TreeCreator in seguito chiama il suo metodo fillArray() una volta per i vertici e una per i nodi, questo metodo ha il compito di impacchettare i dati ricevuti dalla form in 2 array: attributes[] che contiene l'elenco degli attributi e ranges[] che contiene l'elenco dei bound sugli attributi ed in più un flag che indica se il valore desiderato è intero o reale.

In seguito TreeCreator lancia la creazione dei file .csv attraverso i suoi metodi createVertexCSV() e createEdgeCSV().

In seguito viene invocato dall'oggetto TreeConnector il metodo create() di Connector, quest'ultimo istanzia un oggetto GraphAware che viene utilizzato per la connessione al DB.

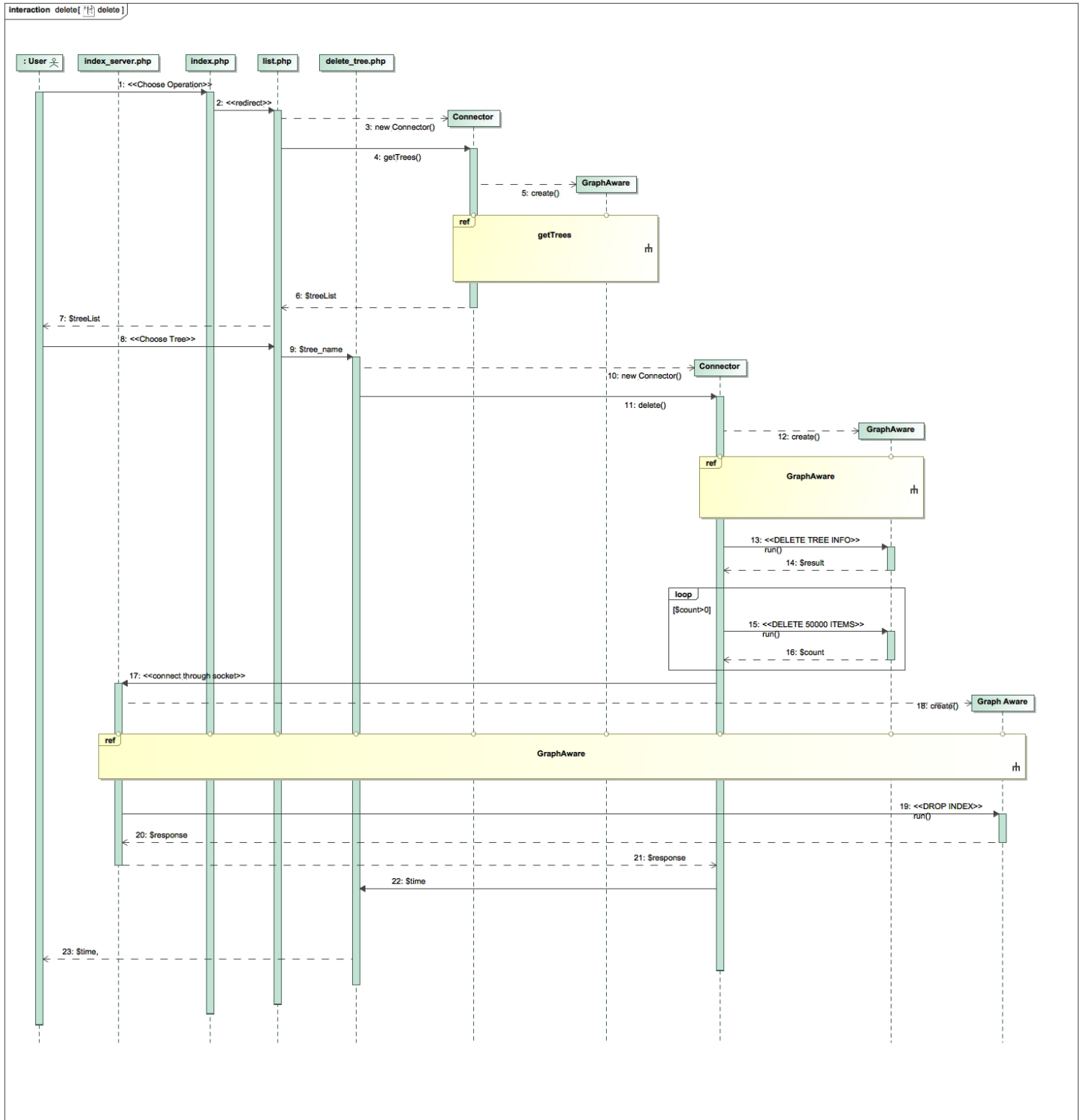
L'oggetto Connector invoca i metodi transaction() e run() passando per argomento la query da somministrare al database per la creazione della struttura contenete i dati relativi ai nodi, il quale tramite GraphAware restituisce \$result (utilizzato come flag di successo).

Dopo la creazione della struttura dei nodi, per evitare deadlock, si usa index_server.php (vedi Design Decision).

La connessione a index_server.php avviene tramite socket, il server (index_server.php) crea quindi una nuova connessione al database istanziando un nuovo oggetto GraphAware, dopo aver stabilito la connessione il server invia la query di creazione degli INDEX, in seguito, dopo aver ricevuto l'ok dal database (attraverso GraphAware), il controllo torna alla classe Connector.

Connector invia poi la query di creazione della struttura relativa ai dati degli archi, infine somministra la query per la creazione del nodo "0" nella struttura Tree Details (vedi Component Diagram) contenente le caratteristiche di ogni albero.

Una volta completato il tutto il controllo viene tornato a TreeCreator che, avendo calcolato il tempo totale di esecuzione lo restituisce alla pagina create_tree.php che lo manda a video all'utente, insieme ad un messaggio di successo.



Sequence Diagram Delete

L'utente dopo aver scelto "Delete" nella pagina `index.php`, verrà reindirizzato alla pagina `list.php`, la quale istanzierà un nuovo oggetto `Connector`, sul quale verrà richiamato il metodo `getTrees()`.

Tale metodo, istanzierà un oggetto `GraphAware` e interroga il database per ottenere la lista degli alberi presenti nel database (per i dettagli vedi sequence diagram `getTrees()`).

La lista di alberi ottenuta verrà restituita a `list.php` che la mette a disposizione dell'utente.

A questo punto l'utente sceglie un albero da eliminare. Il nome di quest'ultimo verrà passato alla pagina `delete_tree.php`, la quale istanzia un nuovo oggetto `Connector`.

`delete_tree.php` invocherà il metodo `delete()` di `Connector`, che istanzia un oggetto `GraphAware` (per la connessione al database; per i dettagli vedi il sequence diagram `GraphAware`).


Si invia una query che elimina il nodo "0" (quello contenente le caratteristiche dell'albero), si esegue poi una serie di cancellazioni progressive in cui vengono eliminati 50000 nodi alla volta e restituisce il numero (count) di nodi eliminati.

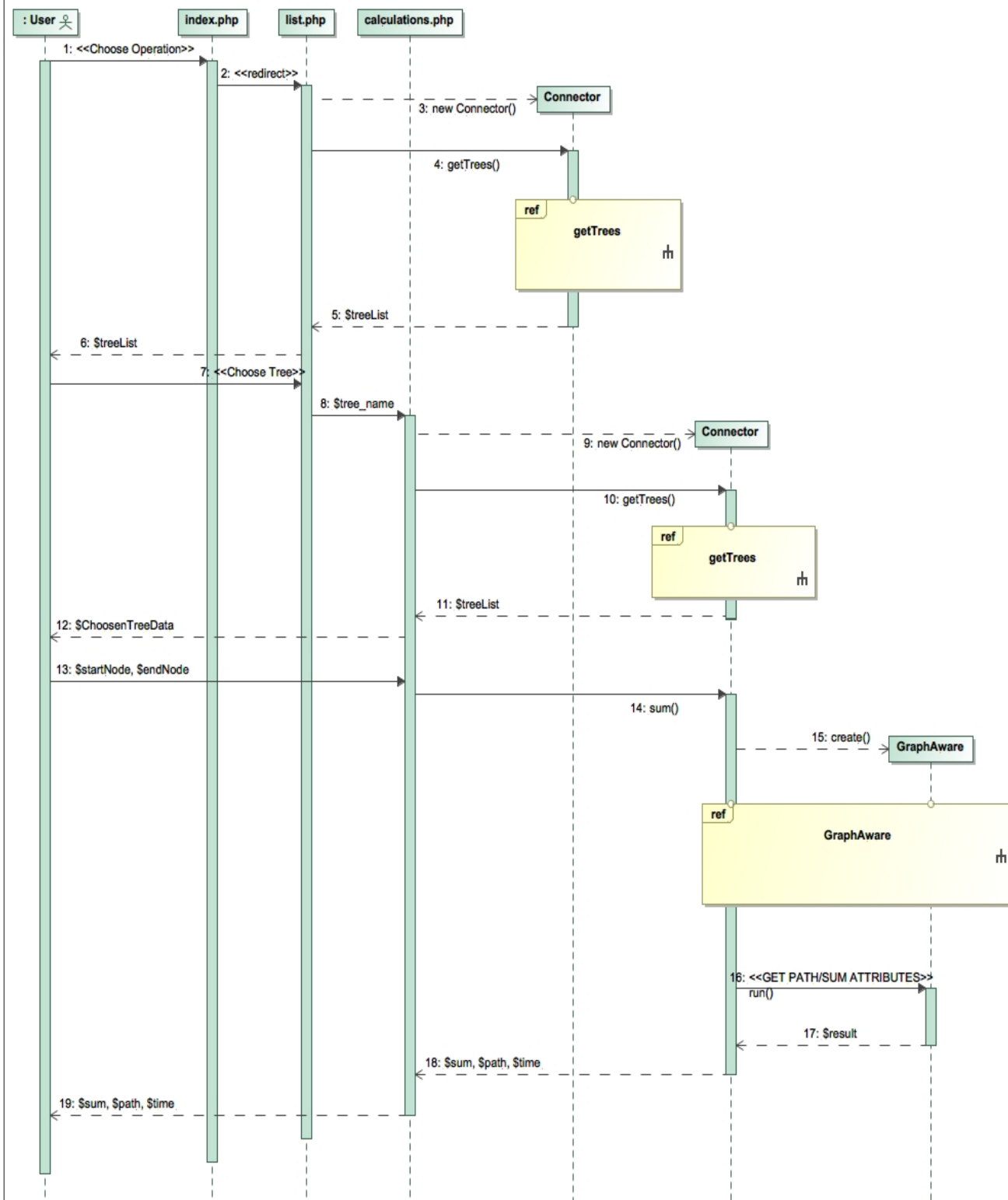
Questa query viene ripetuta finché la variabile `count` non diventa uguale a zero (cioè ha eliminato tutti i nodi dell'albero).

Per eliminare l'INDEX (generato nella fase di creazione), l'oggetto `Connector` richiede una connessione al server (`index_server.php`, per evitare deadlock – vedi design decision), il quale a sua volta istanzia un nuovo oggetto `GraphAware` per la connessione al database e invia la query di `DROP INDEX`.

Una volta completata la cancellazione degli indici, il controllo torna all'oggetto `Connector` che calcola e restituisce a `delete_tree.php` il tempo di esecuzione.

Quest'ultimo viene inviato all'utente (tramite la GUI).

Interaction sum[ sum]



Sequence Diagram Sum

L'utente sceglie l'operazione somma nella pagina index.php, quest'ultima reindirizza alla pagina list.php, la quale istanzia l'oggetto Connector, che attraverso il metodo getTrees() (vedi sequence diagram getTrees) restituirà una treeList (ossia una lista di oggetti di tipo Tree) per permettere all'utente di scegliere l'albero su cui effettuare la somma.

Il nome dell'albero viene inviato alla pagina calculations.php, il quale istanzia un nuovo oggetto Connector che richiama il metodo getTrees(), tramite il quale risale ai dettagli relativi all'albero il cui nome corrisponde a quello passato (vedi sequence diagram getTrees).

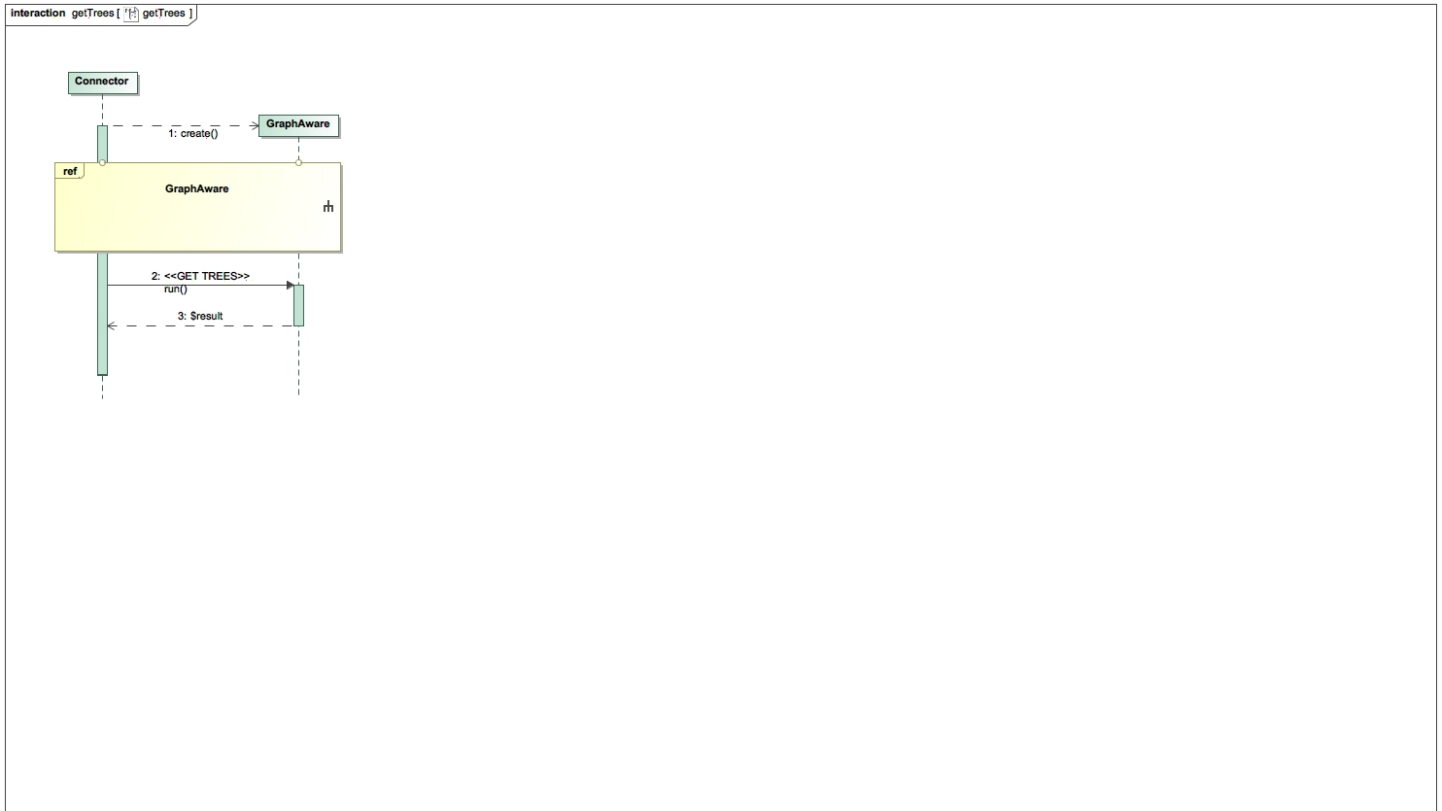
Nuovamente la GUI mette a disposizione dell'utente un'interfaccia dove vengono proposti i dettagli dell'albero scelto e dove può scegliere i Vertici su cui calcolare il path.

I due vertici di inizio e fine path inseriti dall'utente, vengono trasmessi alla pagina calculations.php, la quale richiama il metodo sum() sull'ultimo oggetto Connector istanziato.

L'oggetto Connector, istanzia un nuovo oggetto GraphAware tramite il quale richiede connessione al database(vedi sequence diagram GraphAware).

Ottenuta la connessione lancia la query di somma tramite il metodo run() e ritorna a Connector l'esito della query.

A questo punto, l'oggetto Connector restituisce \$sum, \$path, \$time a calculations.php, che attraverso la Web GUI, mostra all'user finale il risultato della somma, il path, ed il tempo totale impiegato dalla query.

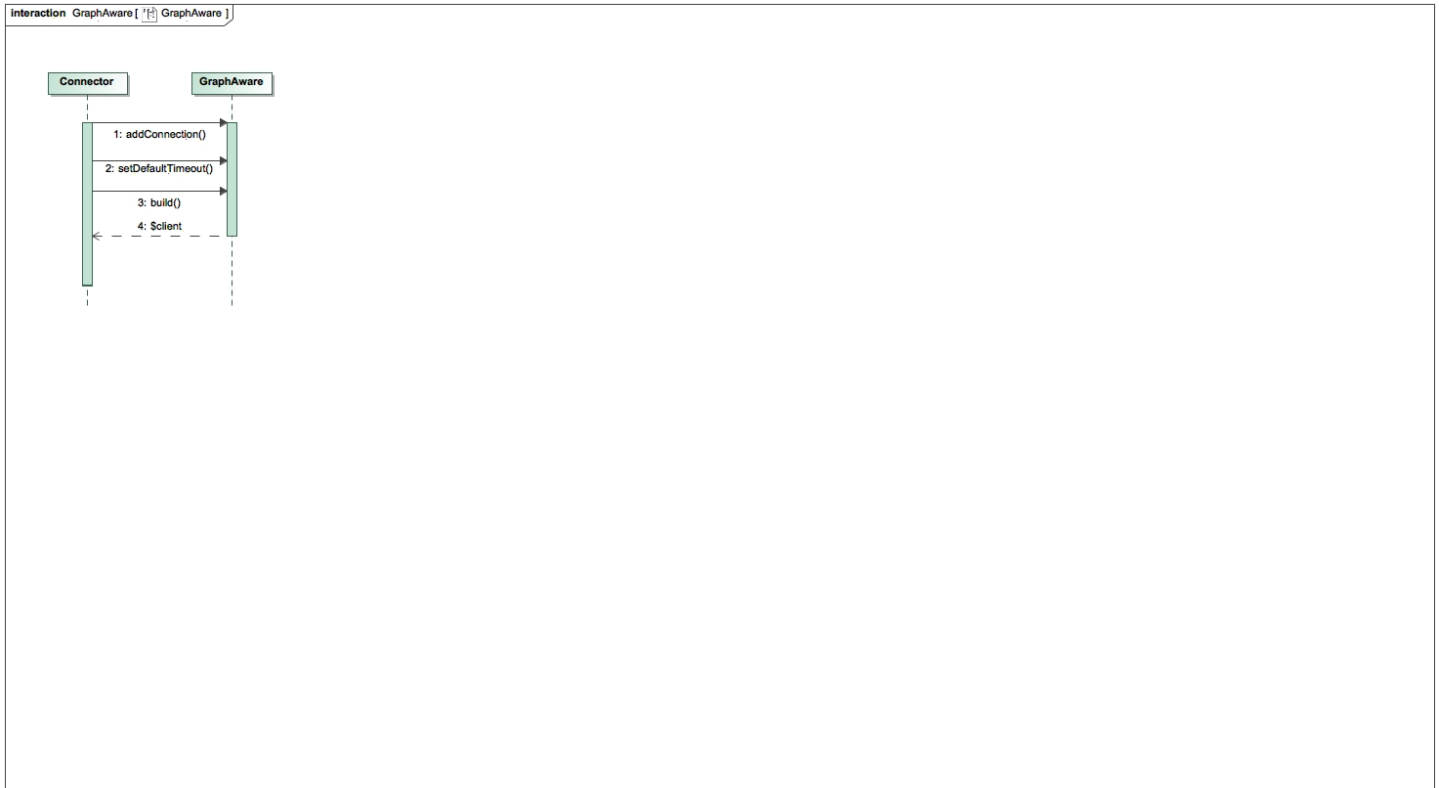


Sequence getTrees

L'oggetto Connector (istanziato precedentemente), crea un oggetto della classe GraphAware che viene utilizzato per la connessione e lo scambio di informazioni con il database.

Una volta che Connector ha fatto richiesta di connessione al database (si veda il sequence diagram GraphAware), esegue il metodo run(), tramite il quale richiede la lista degli alberi presenti nel database.

Tale lista viene ritornata all'oggetto Connector come \$result della chiamata al metodo run().



Sequence Diagram GraphAware

Il seguente sequence diagram mostra la sequenza di operazioni di interfacciamento con Neo4j (tramite le API GraphAware).

La classe Connector richiama i seguenti metodi:

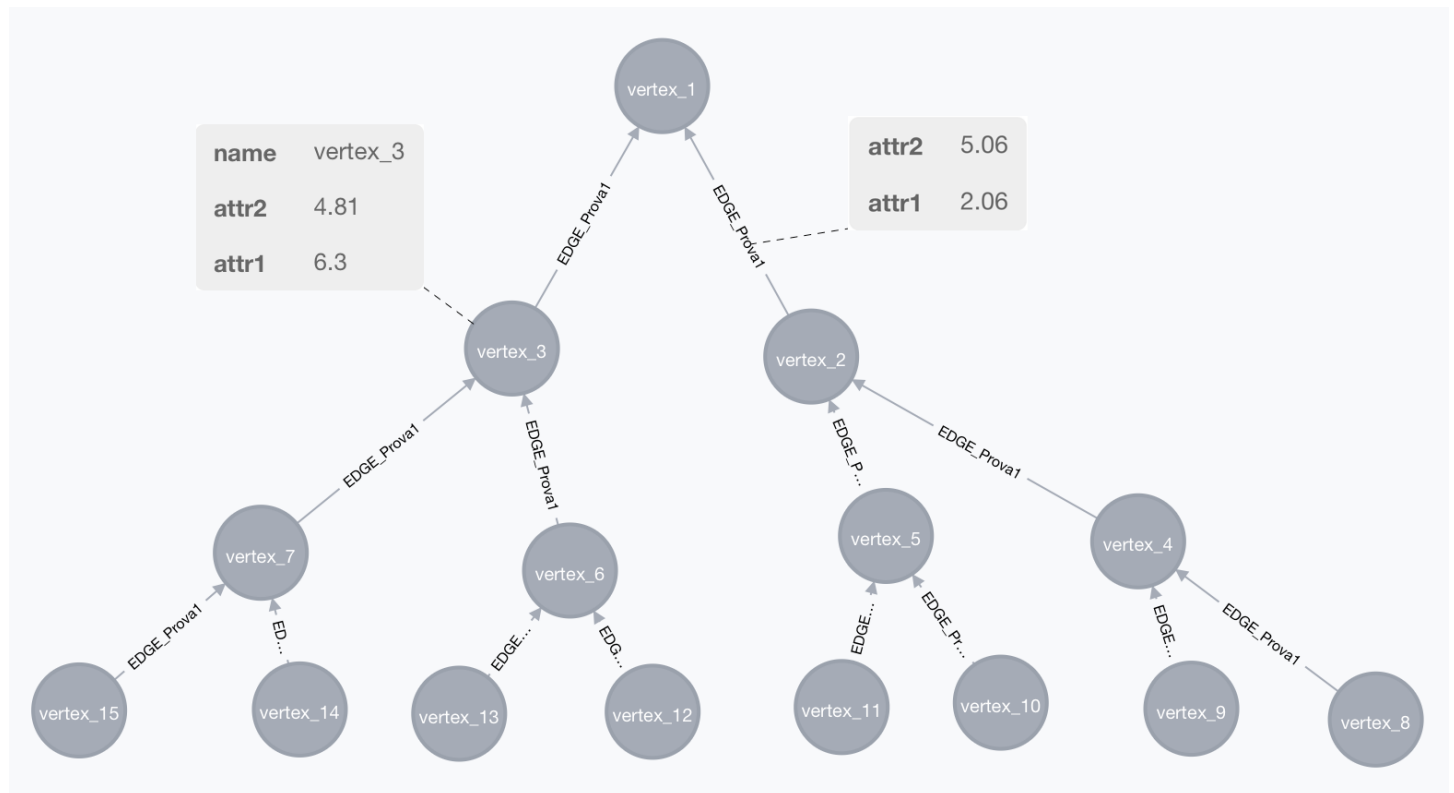
- addConnection(),
- setDefaultTimeout()
- build()

build() in particolare stabilisce la connessione e torna \$client alla classe Connector che verrà successivamente usato come riferimento per l'invio delle query.

D. ER Design

<Report here the Entity Relationship Diagram of the system DB>

Avendo utilizzato un DB NoSQL, per cui il digramma ER non ha un forte potere espressivo, come invece accade per i DB SQL, riportiamo la strutturazione dei dati, sfruttando la strutturazione grafica messa a disposizione da Neo4j.



L'esempio mostra un singolo albero con $\text{split_size} = 2$ e $\text{depth} = 3$ in cui sono stati definiti due attributi sui Vertex (attr1, attr2) e due attributi sugli Edges (attr1, attr2). Nel database questi vengono salvati come proprietà, rispettivamente, dei nodi e degli archi.

Il database conterrà una struttura simile a quella proposta per ogni albero che verrà creato sul database. Neo4j permette di salvare Nodi ed Archi differenziandoli tramite una Label, pertanto la utilizzeremo per suddividere Nodi ed Archi appartenenti ad alberi diversi.

Esempio:

I Nodi dell'albero Prova avranno etichetta Vertex_Prova, gli Archi invece EDGE_Prova;

I Nodi dell'albero Prova1 avranno etichetta Vertex_Prova1, gli Archi invece EDGE_Prova1;

E così via...

L'ultima struttura dati che utilizziamo è un altro tipo di Nodo che "labelliamo" :Tree, di seguito riportiamo come questo tipo di Nodo verrà salvato e quali proprietà conterrà.

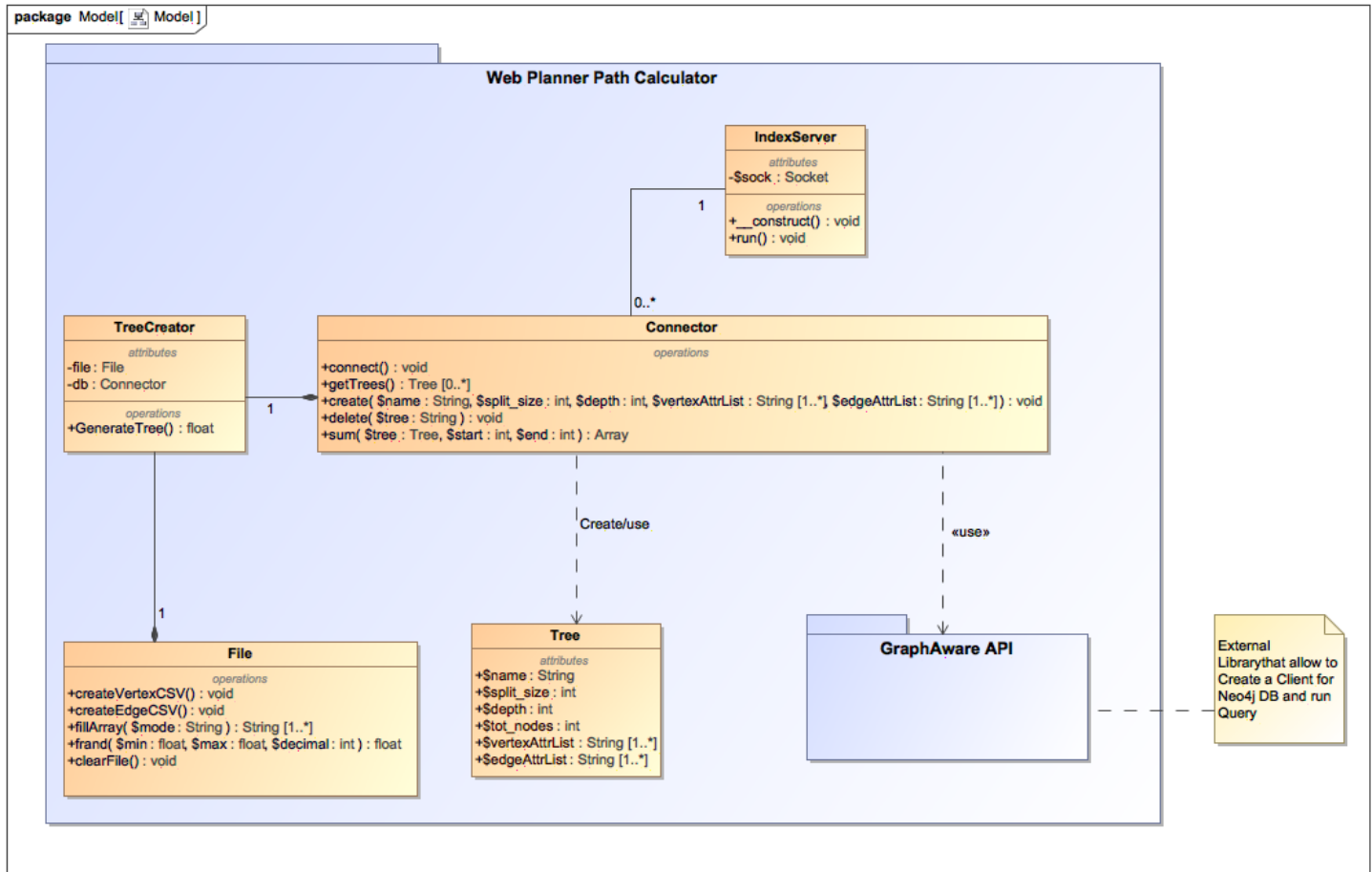
vertexAttrList	[attr1, attr2, attr3, attr4, attr5]
depth	13
split_size	3
name	Prova
edgeAttrList	[attr1, attr2, attr3, attr4, attr5]
tot_nodes	2391484

Verrà generato sul database un nodo simile a questo per ogni albero di cui verrà richiesta la creazione. Esso conterrà informazioni riguardanti l'albero, quali:

- Name;
- Split_size;
- Depth;
- Tot_Nodes;
- VertexAttrList;
- EdgeAttrList.

Queste saranno le informazioni a cui accederà il modulo che dovrà richiedere la Visualizzazione/Selezione di un albero.

E. Class Diagram of the implemented System



Il sistema è formato da cinque classi, “TreeCreator”, “File”, “Tree”, “Connector” e “IndexServer”, e utilizza le API GraphAware.

La classe TreeCreator ha due attributi: file e db. file è un oggetto di tipo File, db è un oggetto di tipo Connector.

La classe ha un unico metodo "GenerateTree()" il quale prima "pulisce" i dati in arrivo dalla form, poi costruisce gli array dei parametri (\$attributes - contenente l'elenco degli attributi e \$ranges - contenente, per ciascun attributo, i bound ai quali sono sottoposti), necessari per la creazione dei file .csv. Infine richiama i metodi per la creazione di nodi e archi (createVertexCSV() e createEdgeCSV()). Il valore di ritorno è un valore float, il quale esprime il tempo impiegato per la creazione.

La classe File ha quattro metodi: "createVertexCSV()", "createEdgeCSV()", "fillArray(\$mode)", "frand(\$min,\$max,\$decimal)" e "clearFile(\$name)".

I metodi "createVertexCSV()" e "createEdgeCSV()" creano rispettivamente i file CSV e non tornano nulla.

Il metodo "fillArray(\$mode)" riempie gli array \$attributes e \$ranges con i valori ricevuti in POST dalla form, distinguendo fra vertici e nodi grazie al parametro "\$mode", in quanto tali array avranno strutture diverse. Restituiscono gli array generati.

Il metodo "frand(\$min,\$max,\$decimal)" genera e restituisce un valore random, il quale sarà sempre compreso fra il parametro \$min e \$max.

Il numero di decimali è espresso dal parametro \$decimal, che, se settato a 0, restituirà un valore random intero.

La classe Connector ha cinque metodi: "connect()", "getTrees()", "create(\$name, \$split_size, \$depth, \$vertexAttrList, \$edgeAttrList)", "delete(\$tree)" e "sum(\$tree, \$start, \$end)".

Il metodo "connect()" instaura una connessione con il DB, tramite le API GraphAware. Non ha nessun valore di ritorno.

Il metodo "getTrees()" recupera sul DB la lista degli alberi (con i loro attributi) e la restituisce sotto forma di array di oggetti tree.

Il metodo "create(\$name, \$split_size, \$depth, \$vertexAttrList, \$edgeAttrList)" presi in input: il nome, la split_size, la depth e le relative liste di attributi lancerà le query di creazione sul DB tramite le API GraphAware. Non ha nessun valore di ritorno.

Il metodo "delete(\$tree)", preso come parametro il nome dell'albero invia al DB le query di cancellazione. Non ha nessun valore di ritorno.

Il metodo "sum(\$tree, \$start, \$end)", presi un oggetto di tipo tree, il nodo di partenza e di arrivo, invia la query di ricerca del path tra il nodo \$start e il nodo \$end, la stessa query è arricchita con direttive atte al calcolo della somma sugli elementi appartenenti al path. Restituisce un array contenente i risultati.

La classe Tree ha sei attributi: \$name indica il nome dell'albero, \$split_size indica il numero di figli di ogni nodo, \$depth indica la profondità dell'albero, \$tot_nodes indica il totale dei nodi dell'albero, \$vertexAttrList e \$edgeAttrList indicano le liste degli attributi dei nodi e degli archi, rispettivamente.

GraphAware è una libreria esterna che instaura la comunicazione tra php e Neo4j permettendo di somministrare query e ottenerne risultati.

La classe TreeCreator ha una relazione di composizione con classe File e la classe Connector: nel momento in cui verrà creata un'istanza della classe TreeCreator verranno create anche le istanze di File e Connector, le quali faranno parte esclusivamente dell'oggetto TreeCreator appena creato.

La classe Connector, in base ai casi, usa o crea un'istanza della classe Tree.

La classe Connector usa le API GraphAware.

La classe **IndexServer** è la classe atta alla creazione dei socket utilizzati per gestire e risolvere il problema delle concorrenze, così da evitare che il sistema possa andare in stato di deadlock. Ha un solo attributo \$sock che è un oggetto di tipo socket ed è composta di due metodi:

- __construct(): è il costruttore della classe e si occupa della costruzione del socket.
- run(): è il metodo che attiva le funzioni del socket.

F. Design Decisions

<Document here the 5 most important design decisions you had to take. You can use both a textual or a diagrammatic specification.>

Non Modifica dell'albero

L'albero una volta creato non potrà essere modificato. Era molto importante sapere se la modifica fosse una funzionalità necessaria per il sistema perché avrebbe influito particolarmente sulle nostre scelte. Il committente ci ha riferito che la modifica non era una funzionalità richiesta dal Sistema, quindi abbiamo deciso di non implementarla per rendere il nostro sistema più performante:

Non essendoci scritture concorrenziali il sistema potrà permettere senza problemi la visualizzazione e la richiesta delle operazioni di somma su uno stesso albero a diversi utenti in parallelo, non incappando nel problema di dati inconsistenti.

Visualizzazione dell'albero

La visualizzazione dell'albero è risultata ambigua e ci è sembrato opportuno decidere cosa avrebbe significato per il nostro Sistema, le possibilità che ci si presentavano sono le seguenti:

1. Visualizzazione Grafica dell'intero albero (messo a disposizione dal database Neo4j):

Il problema di questa soluzione sono le performance, andremo a trattare con alberi di milioni di nodi, doverli recuperare dal database e metterli a video sarebbe dispendioso dal punto di vista delle performace (richiederebbe del tempo, sovraccaricando il DB) e soprattutto poco user-friendly; la GUI risulterebbe troppo "pesante".

2. Visualizzazione delle Proprietà principali dell'albero:

Verranno visualizzati nella GUI, una volta scelto un albero, le sue informazioni principali utili al suo impiego:

- Nome dell'albero;
- Split Size;
- Depth;
- AttributeVertexList;
- AttributeEdgeList;
- Numero totale di Vertici.

Ci è sembrato più opportuno scegliere la seconda opzione in quanto più pratica ed efficiente.

Per implementare questa operazione e renderla semplice e veloce, l'idea è quella di salvare tutte le proprietà precedentemente citate relative ad un albero in un nodo fittizio. La visualizzazione pertanto consisterà nel recuperare questo nodo (spetterà all'Engine interrogare il Database) dopodiché la GUI mostrerà le informazioni relative all'albero.

Attributi

I valori degli attributi saranno generati solo in maniera random tra un range K – N inseriti da input. L'albero potrebbe arrivare ad avere un numero elevatissimo di attributi e per lo scopo per cui è stato idealizzato il Sistema, sarebbe dispersivo e poco funzionale dare la possibilità di inserirli manualmente.

DBMS

Come spiegato anche nell'analisi dei rischi, la scelta del database è stata un punto di discussione. La nostra scelta è ricaduta su un Database a Grafo NoSQL, Neo4j. Quest'ultimo è open-source e quindi soddisfa i vincoli imposti dal committente, inoltre come precedentemente detto fornisce gratuitamente l'implementazione dei grafi e delle operazioni più importanti su di essi (es. Calcolo di un Path tra due nodi). Per quanto riguarda le performance sembra rientrare in maniera molto larga nei vincoli di tempo richiesti.

Il database permette esclusivamente di salvare dati strutturandoli come Nodi ed Archi, con la possibilità di definire degli attributi su di essi. Quindi salveremo realmente sul database le strutture ad albero vere e proprie.

I primi test che abbiamo effettuato per la CREAZIONE della struttura ad albero mostravano dei tempi poco accettabili, quindi abbiamo dovuto trovare delle modalità di creazione più performanti:

- L'albero verrà creato dando come input al Database due file .csv creati dall'Engine, uno per i Nodi ed uno per gli Archi, in cui ogni riga rappresenterà un elemento (nodo o arco) da creare. La query è la seguente:

```
Using periodic commit 50000 LOAD CSV WITH HEADERS FROM 'file:///Nodi.csv' AS line
Create(:Vertex_nomealbero {name:line.name,
attr1: toFloat(line.attr1), attr2: toFloat(line.attr2),
attr3: toFloat(line.attr3), attr4: toFloat(line.attr4),
attr5: toFloat(line.attr5)})
```

L'esempio proposto mostra la creazione da file .csv dei nodi. Il collo di bottiglia della creazione è risultato essere la COMMIT: il DB dovendo fare una Transaction per la creazione di ogni singolo nodo impiegava troppo tempo. Neo4j durante l'"import" da file .csv, permette di forzare la Commit ogni X letture di righe (nell'esempio X = 50.000); Questo ha provveduto ad abbassare notevolmente la creazione dei soli nodi in un tempo di circa 1 minuto.

Altro problema relativo alle performance:

- In una qualsiasi altra query (creazione archi, somma...) si presenterà il problema di “ritrovare” tra i milioni di nodi di un certo albero dei nodi in particolare (per creare un arco dal nodo A al nodo B dovremmo prima ritrovare il nodo A e il nodo B). Quello che Neo4j mette a disposizione per migliorare questo aspetto è la creazione degli index su un determinato tipo di nodo e su determinati attributi di quel tipo di nodo. L'index di Neo4j è molto simile a quelli messi a disposizione dai DB SQL e consiste detto in parole povere in una “copia ridondante” dei dati per cui si è richiesta la creazione dell'index.

```
Create index on :Vertex_nomealbero(name)
```

Così facendo d'ora in poi, ogniqualevolta si richiederà la ricerca di un nodo, Neo4j utilizzerà l'indice per velocizzare l'operazione sui vertici di tipo :Vertex_nomealbero sull'attributo name. Suddetta operazione andrà eseguita ogni volta si creeranno dei nuovi tipi di nodi. Dato che per lo spazio occupato dal Sistema non è stato imposto un vincolo molto stretto, abbiamo scelto di impiegare più spazio (quello richiesto per l'index) per migliorare quello per cui ci sono stati imposti vincoli più importanti, ovvero le performance.

Operazione di Somma

L'operazione fulcro del sistema, quella per cui ci sono stati imposti vincoli più stretti è quella anche per cui abbiamo discusso maggiormente: ci ha portati a scegliere il database a Grafo così da avere gratuitamente le operazioni più comuni sui grafi (come quella di trovare un cammino tra due vertici) già implementate. Altra decisione chiave è: “Dove andrà eseguita la somma?”, “Sarà l'Engine ad effettuarla, oppure il Database?”. Abbiamo ritenuto necessario effettuare dei primi test su degli alberi creati sul Database per constatare effettivamente se l'operazione di somma rientrava nei vincoli espressi:

- Split Size = 3 e Depth = 13, ca. 2.400.000 nodi: un tempo che varia tra i 18 e i 6 secondi;
- Split Size = 2 e Depth = 20, ca. 2.000.000 nodi: un tempo inferiore al decimo di secondo (>100 ms);

In entrambi i casi parliamo di 10 Attributi sui Vertici e 5 sugli Archi, tali prove sono state effettuate direttamente sul Database.

Pertanto abbiamo deciso, allo stato dell'arte, che la somma verrà effettuata direttamente sul database tramite una Query (che verrà inviata dall'Engine al DB stesso) che tornerà:

- i. Per ogni attributo, la somma di tutti gli attributi dal nodo di partenza al nodo di destinazione;
- ii. La lista dei vertici appartenenti al cammino dal nodo di partenza al nodo di destinazione.

Questi risultati verranno poi elaborati dall'Engine per essere poi passati alla GUI che li mostrerà come risultato all'utente; inoltre provvederà a calcolare il tempo richiesto per tale operazione.

La “curiosità” che non ci convinceva era questa:

L’operazione di somma è legata, computazionalmente, all’altezza dell’albero (= lunghezza massima del cammino), quindi, ragionevolmente, ci si aspetta che in un albero più alto che la computazione richieda più tempo. Quello che invece i nostri test mostravano era completamente l’opposto (pur rientrando nei tempi richiesti). Dopo averci speso del tempo per indagare su ciò, abbiamo trovato la radice del problema: come spiegato in precedenza abbiamo creato degli indici sui vertici di tipo :Vertex_nomealbero(name), questi dovrebbero venire usati in automatico in Neo4j quando il DBMS lo ritiene necessario. Evidentemente per l’albero di altezza 20 lo ritiene necessario e lo utilizza, mentre per l’albero di altezza 13 no. Quindi abbiamo dovuto trovare un modo per forzare l’utilizzo dell’index.

```
Match
p=(from:Vertex_ss3 {name:'vertex_2391484'})-[:EDGE_ss3*]->(to:Vertex_ss3 {name:'vertex_1'})
using index from:Vertex_ss3(name)
using index to:Vertex_ss3(name)
with nodes(p) as nod, relationships(p) as rels
return
nod,
reduce(sum = 0, n IN nod| sum + n.attr1) as tot_attr1_node,
reduce(sum = 0, n IN nod| sum + n.attr2) as tot_attr2_node,
reduce(sum = 0, n IN nod| sum + n.attr3) as tot_attr3_node,
reduce(sum = 0, n IN nod| sum + n.attr4) as tot_attr4_node,
reduce(sum = 0, n IN nod| sum + n.attr5) as tot_attr5_node,
reduce(sum = 0, n IN nod| sum + n.attr6) as tot_attr6_node,
reduce(sum = 0, n IN nod| sum + n.attr7) as tot_attr7_node,
reduce(sum = 0, n IN nod| sum + n.attr8) as tot_attr8_node,
reduce(sum = 0, n IN nod| sum + n.attr9) as tot_attr9_node,
reduce(sum = 0, n IN nod| sum + n.attr10) as tot_attr10_node,
reduce(sum = 0, n IN rels| sum + n.attr1) as tot_attr1_edge,
reduce(sum = 0, n IN rels| sum + n.attr2) as tot_attr2_edge,
reduce(sum = 0, n IN rels| sum + n.attr3) as tot_attr3_edge,
```

Il codice qui presentato mostra la query da noi studiata per effettuare la somma sull’albero, le due righe evidenziate in rosso sono state aggiunte successivamente per forzare l’uso dell’indice sul vertice from:Vertex_ss3(name) e sul vertice to:Vertex_ss3(name). Usando questa Query per la somma i tempi non superano i 100ms.

Creazione dell’albero

Come espresso nella tabella delle Challenge ci si sono posti problemi nel momento in cui abbiamo lavorato per gestire la concorrenza. Il Database, nel creare gli Index in parallelo per ogni albero per cui gli viene richiesta la creazione, aveva la possibilità di andare in Deadlock. Il problema era quindi serializzare la creazione degli Index (che singolarmente richiedono un tempo esiguo) evitando così tale situazione critica. A tal proposito abbiamo realizzato un servizio di Server/Client in cui i Client sono i vari processi generati dalle richieste di creazione provenienti dagli utenti, mentre il Server sarà il delegato alla creazione/cancellazione degli Indici.

I diversi Client delegheranno al DB la creazione dei nodi e degli archi del relativo albero, ma fra queste due operazioni, manderanno una richiesta al Server tramite Socket per la suddetta creazione degli Indici in maniera sincrona, aspettando un “acknowledge” dal Server in caso di corretta creazione dell’Indice che gli permetterà di continuare il loro lavoro di creazione.

Il Server, rimarrà in continuo ascolto (in background) su un Socket, non appena arriverà una richiesta di creazione/cancellazione di Indice, eseguirà tale operazione sul database. Mentre il Server è occupato alla risoluzione di una richiesta, accoderà le richieste in arrivo e man mano verranno evase una alla volta.

Come detto l’operazione di Creazione degli Index è un’operazione critica, ma non richiede molto tempo di esecuzione, infatti con l’inserimento di questo sistema non si è notato un incremento del tempo totale di creazione degno di nota.

G. Explain how the FRs and the NFRs are satisfied by design

<Report in this section how the design you produced satisfies the FRs and the NFRs>

GUI Requirements

La GUI è basata su HTML5, permette un'immediata interazione con l'utente esponendo le funzionalità del sistema attraverso un menù laterale.

La GUI dà la possibilità, in fase di creazione, di indicare le varie caratteristiche dell'albero e permette, in modo dinamico, l'aggiunta di un numero arbitrario di attributi sui nodi e sugli archi.

Per ogni attributo sarà possibile indicare i bound entro i quali verrà generato il valore randomico, inoltre, permette di specificare se si vuole che il valore generato sia intero o reale.

Nella prossima revisione abbiamo intenzione di permettere di indicare anche il nome dell'attributo (questa possibilità è stata aggiunta nell'ultima versione).

La GUI permette, prima di procedere con le operazioni di cancellazione e di somma, di visualizzare l'elenco degli alberi precedentemente creati insieme alle loro caratteristiche.

Una volta selezionato l'albero sarà possibile indicare un nodo di partenza e uno di arrivo per calcolare il cammino e le somme sugli attributi dei nodi e degli archi appartenenti allo stesso, oppure si può procedere all'eliminazione.

Ogni operazione (Creazione, Cancellazione e Somma) forniscono, oltre ai vari output, anche il tempo impiegato per l'esecuzione espresso in secondi.

Business Logic Requirements

Il sistema è in grado di modellare grafi in generale, nello specifico alberi n-ari come richiesto dai requisiti.

In fase di creazione, attraverso i dati inseriti nella GUI il sistema genera un valore randomico attraverso la funzione `frand()`.

L'Engine permette la costruzione (in base ai parametri specificati) e la somministrazione di query al DB.

L'Engine, prima dell'invio delle query nel caso della creazione genera 2 file .csv per ogni albero contenenti i dati relativi a nodi e archi.

Ognuno di questi file, nel caso di 3 attributi su nodi e archi, ha un peso di circa 60MB.

A parte i controlli sui dati inseriti nelle form e la pulizia degli stessi da caratteri speciali non vengono implementati ulteriori accorgimenti per quanto riguarda la security del sistema.

DB Requirements

Il database NoSQL Neo4j permette di salvare gli alberi nella modalità descritta nel punto **D**.

L'operazione di somma avviene in pochi decimi di secondo (anche se ne vengono eseguite diverse in parallelo e mentre vengono fatte altre operazioni sul db), per il nostro sistema la somma è valida anche nel caso in cui si perda la proprietà dell'orientamento degli archi, grazie alla capacità intrinseca del DB di trovare il path tra due nodi.

Una osservazione sulle modalità di caching di Neo4j:

Il DBMS stesso ha riservato per sé una porzione di memoria RAM da utilizzare per mantenere i dati delle query che più frequentemente vengono richiesti. Quindi con il Sistema "a caldo" si avrà la certezza di avere in RAM i dati degli alberi più frequentemente richiesti per la somma, incrementando ancora di più le performance della somma su tali alberi.

E' stata aggiunta la possibilità di scegliere in fase di somma di considerare l'albero come orientato oppure no.

L'operazione di creazione impiega circa 3-4 minuti su singolo albero e circa 30 minuti quando abbiamo creazioni multiple simultanee (nei nostri test 12 alberi, tutti da 2 milioni di nodi - split size 2 e depth 20, la cui creazione è stata richiesta nello stesso istante).

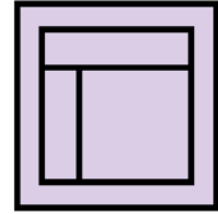
15 creazioni simultanee sono considerate il caso pessimo: abbiamo 1 creazione a settimana per utente, quindi al peggio 100 creazioni a settimana che corrispondono a circa 15 creazioni al giorno.

E' statisticamente improbabile che avvengano 12-15 creazioni di alberi da 2 milioni di nodi nello stesso momento, ma comunque, anche in quel caso, il sistema rispetta i requisiti di performance imposti.

Il DB permette una buona scalabilità, in particolare le concorrenze (in scrittura e in cancellazione) sono state risolte tramite l'utilizzo di un server socket che mette in coda l'esecuzione delle query che potrebbero portare al deadlock (vedi Design Decision).

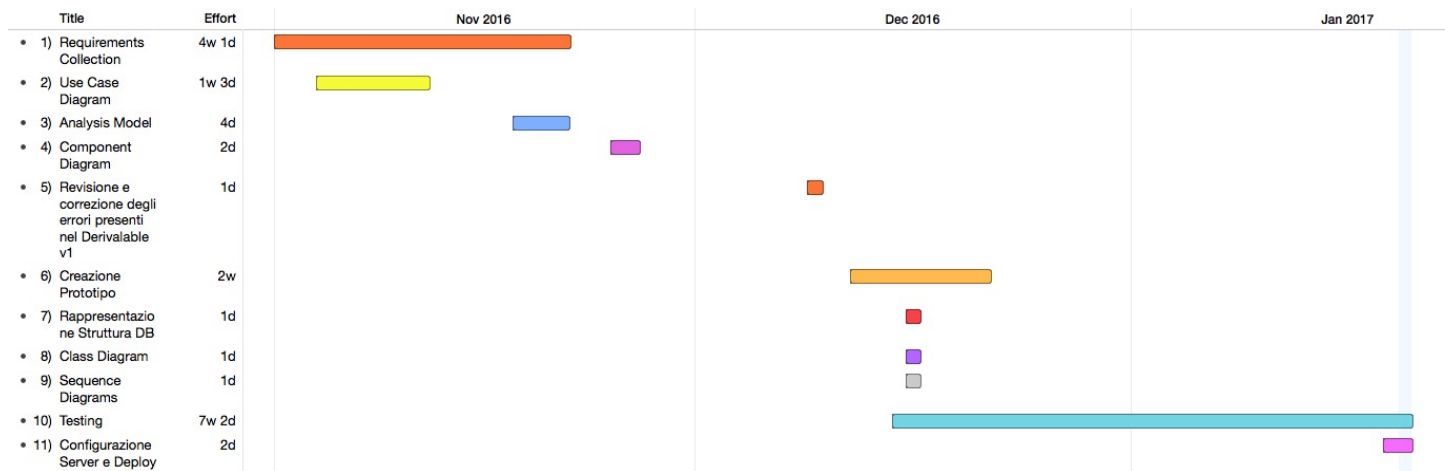
Per quanto riguarda la somma non ci sono conflitti in quanto abbiamo solo letture.

H. Effort Recording



GANTT

Make a GANTT documenting the tasks and timing you expect to spend on the deliverable. Try to be as precise as possible. Check, after the deliverable deadline, if and how you satisfied (or not) the deadlines.



Logging

As you are working on the assignment, record what you are doing and how long you spent. As a rule of thumb, you should add a log entry every time you switch tasks. For example, if you do something for two hours straight, that can be one log entry. However, if you do two or three things in half an hour, you must have a log entry for each of them. You do not need to include time for logging, but should include the time spent answering the other parts of this question.

For this purpose, please use the **LogTemplate.xls** file.

Categorization

When logging the time spent on the project, please create different sub-categories. Specifically, it is important to clearly distinguish between two main categories: the time spent for “**learning**” (the modeling languages, the tools, etc.) from the time needed for “**doing**” (creating the models, taking the decisions, ...). Learning tasks are in fact costs to be paid only once, while doing costs are those that will be repeated through the project.

For each category, please define sub-categories. Examples follow. You may add other sub-categories you find useful.

Learning

- Requirements Engineering
- Non functional Requirements
- Use Case Diagrams
- Tool study

Doing:

- Requirements discovery
- Requirements Modeling (UC diagrams)

Summary Statistics

Based on the attributes defined above, calculate the summary statistics of the time spent for “learning”, the time spent for “doing”, and the total time.

Note: this Deliverable report shall document only the Summary Statistics for the different deliverables (D1, D2, and Final). Detailed information shall be reported in the Excel file.

Ore totali Deliverable v1: 68

Ore totali Deliverable v2: 20.5

Ore totali Deliverable v3: 21

COPY HERE (computed from the spreadsheet): i) the total number of hours spent by the group (that is, hours per task X number of people working on that task), ii) the time spent for LEARNING and for DOING

Appendix. Code

<Report in this section a **documented** version of the produced code>

Codice sorgente disponibile nell'allegato alla mail di consegna, nell'indice di questo Deliverable è presente un video dimostrativo ed il link all'applicazione.