



UNIVERSITA' DEGLI STUDI DI FIRENZE

Scuola di Ingegneria - Dipartimento di Ingegneria
dell'Informazione

Corso di Laurea in Ingegneria del Software

“Refactoring di Applicazioni Monolitiche”

Obiettivo e Descrizione	3
Requisiti	3
Use Case	4
Use Case Template	5
Soluzione Proposta	8
INPUT	8
SVOLGIMENTO	8
Modellazione del domain model come un grafo pesato	8
Semplificazione del grafo	9
Progettazione della soluzione	11
Class Diagram	11
ApplicationModel	11
CoOccurrenceMatrix	11
ApplicationAbstraction	12
Weighted Graph	13
Design Patterns	14
STRATEGY	14
FACTORY	16
Loss Functions	18
SimplifyGraphStrategy	20
GraphManager	21
Aggiornamento Use Case Diagram	22
InputManager	22
Mockups	23
Possibili aggiunte	27
Implementazione	28
Classi ed Interfacce	28

CoOccurrenceMatrix	28
ApplicationAbstraction, EndPoint, UseCase e BusinessLogic	29
Coupling	32
DomainModel	33
BuildCoMatStrategy e BuildMatricesFactory	33
GraphManager	34
SimplifyGraphStrategy, SimplifyGraphFactory, LossFunctionStrategy e LossFunctionFactory	36
Graph, Edge e Vertex	38
Unit Test	41
AppModelTest	41
WeightedGraphTest	45
ManagerTest	48
Appendice	54
Come usare il programma	54
Sequence Diagram	56
Bibliografia	57

Obiettivo e Descrizione

Si vuole effettuare il Refactoring di una applicazione monolitica Restful in forma di microservizi.

Un'applicazione monolitica Restful è dotata di un unico Layer e di un unico Domain Model su cui si espongono tutti i servizi.

Un'architettura basata su microservizi è realizzata da componenti indipendenti che eseguono ciascun processo applicativo come un servizio.

Si parte quindi da un Input che rappresenta la descrizione di un'Applicazione Monolitica, e tramite opportuni calcoli, descritti di seguito, otterremo la stessa Applicazione, quindi con le stesse funzionalità, ma basata su una struttura a Microservizi.

Requisiti

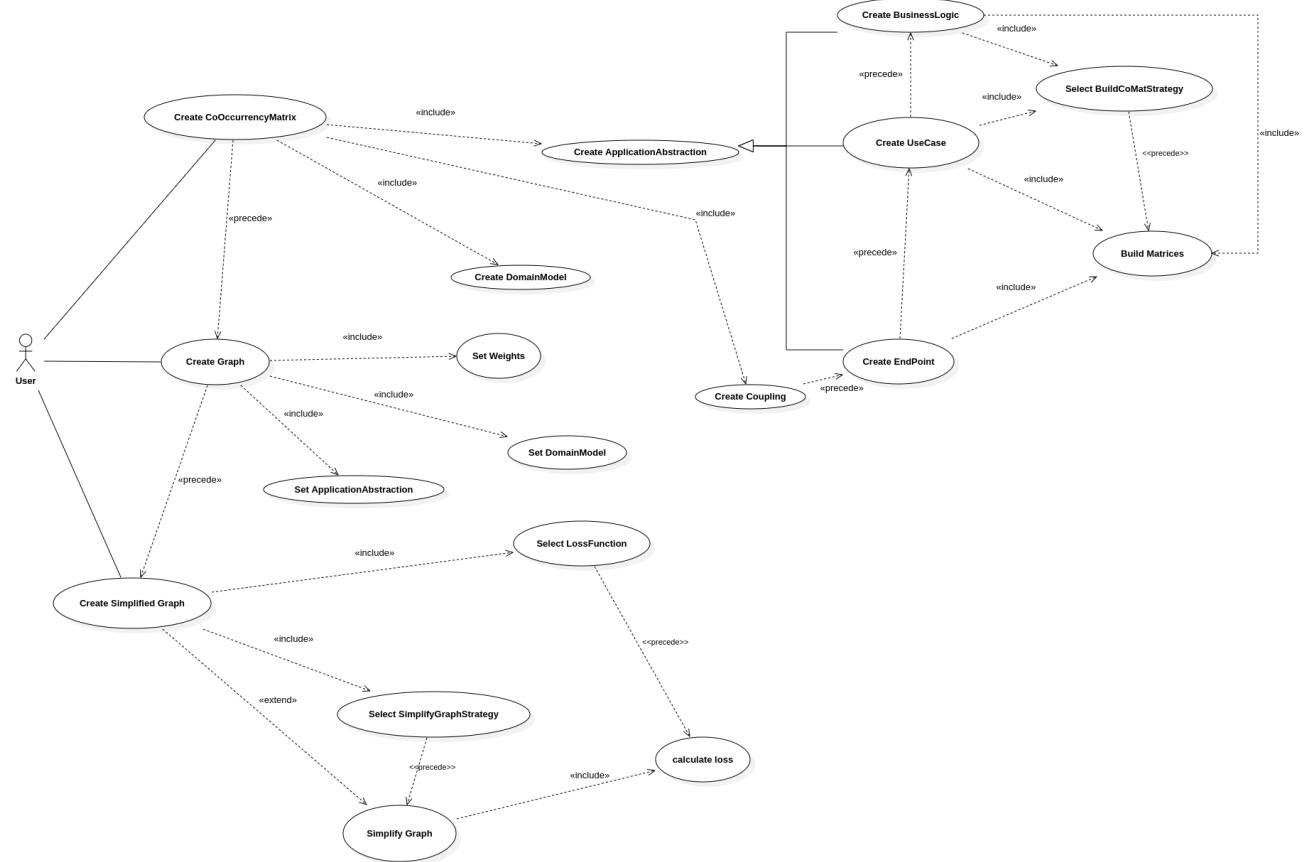
Alcuni requisiti importanti per la nostra applicazione sono:

- **Espandibilità:**
 - Fornire all'utente la possibilità di scegliere il metodo con cui costruire le matrici di co-occorrenza.
 - Fornire all'utente la possibilità di scegliere la strategia di semplificazione del grafo.
 - Fornire all'utente la possibilità di scegliere la funzione di valutazione delle soluzioni.

Le strategie di semplificazione potrebbero avere parametri diversi.

- **Usabilità:** rendere l'applicazione User Friendly in modo che sia facile da usare e da capire, soprattutto in questa fase in cui non è prevista una GUI.
- **Efficacia:** che risolva il problema specificato di suddivisione di una Applicazione Monolitica in Microservizi.
- **Compatibilità:** Utilizzando il linguaggio Java, l'applicazione potrà essere utilizzata su Sistemi Operativi eterogenei.

Use Case



Come si può notare dal diagramma sopra, l'utente, dal main, effettua innanzitutto la **creazione delle matrici di co-occorrenza**, operazione necessaria per poter proseguire con la **creazione del grafo**.

Dopo aver svolto queste due fasi, l'utente, sempre dal main, può invocare il metodo Simplify Graph, il quale **semplifica il grafo** utilizzando come metodo di semplificazione quello specificato dall'utente tramite input. Il metodo calcolerà anche la loss.

Use Case Template

ID:	UC1
Title:	Create Co-Occurrence Matrices
Description:	L'utente vuole costruire le matrici di co-occorrenza
Primary Actor:	User
Preconditions:	<p>L'utente deve possedere un file contenente:</p> <ul style="list-style-type: none"> ● Nomi delle entità ● Descrizione Coupling: <ul style="list-style-type: none"> ○ Nome, entità che ne fanno parte, tipo, valore ● Descrizione degli endpoint: <ul style="list-style-type: none"> ○ Nome, coupling che ne fanno parte. ● Descrizione degli use case: <ul style="list-style-type: none"> ○ Nome, endpoint che ne fanno parte. ● Nome business logic.
Postconditions:	Nell'oggetto di tipo BusinessLogic ci saranno le matrici di co-occorrenza correttamente create.
Main Success Scenario:	<ol style="list-style-type: none"> 1. Creazione oggetto di tipo DomainModel 2. Si riceve in Input il nome delle varie entità: <ol style="list-style-type: none"> 2.1. Creazione oggetti di tipo Entity 2.2. Aggiunta oggetti di tipo Entity all'oggetto di tipo DomainModel 3. Si riceve in Input la descrizione degli end point: <ol style="list-style-type: none"> 3.1. Creazione oggetto/i di tipo EndPoint 4. Si riceve in Input descrizione dei coupling: <ol style="list-style-type: none"> 4.1. Creazione oggetti di tipo Coupling 4.2. Aggiunta oggetti di tipo Coupling all'EndPoint 4.3. Creazione matrici di co-occorrenza per ogni oggetto di tipo EndPoint: 5. Si riceve in Input la descrizione degli use case: <ol style="list-style-type: none"> 5.1. Creazione oggetto/i di tipo UseCase 5.2. Aggiunta oggetti di tipo EndPoint ai vari UseCase 5.3. Scelta strategia per creare le matrici di co-occorrenza (all'interno dell'oggetto di tipo UseCase) 5.4. Creazione matrici di co-occorrenza (all'interno dell'oggetto di tipo useCase) 6. Creazione Oggetto BusinessLogic <ol style="list-style-type: none"> 6.1. Aggiunta oggetti di tipo UseCase 6.2. scelta strategia per creare le matrici di co-occorrenza (all'interno dell'oggetto BusinessLogic) 6.3. Creazione matrici di co-occorrenza (all'interno dell'oggetto di tipo BusinessLogic)
Extensions:	<p>l'utente può decidere di usare le matrici di co-occorrenza di un oggetto di tipo UseCase: fermandosi al punto 5.4.</p> <p>l'utente può decidere di usare le matrici di co-occorrenza di un singolo oggetto di tipo EndPoint: si ferma al punto 4.3.</p> <p>In caso di oggetti di tipo Coupling con stesse Entità (con stesso ordine) e stesso tipo, ma con diverso valore di accoppiamento, buildMatrices fallisce.</p> <p>Nelle varie liste di oggetti di tipo Coupling in oggetti di tipo EndPoint, o liste di oggetti di tipo EndPoint in oggetti di tipo UseCase, o lista di oggetti di tipo UseCase in un oggetto di tipo BusinessLogic: i doppioni verranno scartati interpretandoli come errori di input.</p>

ID:	UC2
Title:	Create Graph
Description:	L'utente vuole creare un Grafo
Primary Actor:	User
Preconditions:	L'utente deve instanziare : <ul style="list-style-type: none"> • Un oggetto di tipo ApplicationAbstraction con anche le sue matrici di co-occorrenza istanziate.
Postconditions:	Nell'oggetto di tipo GraphManager sarà salvato il grafo.
Main Success Scenario:	<ol style="list-style-type: none"> 1. Creazione oggetto di tipo GraphMaganer <ol style="list-style-type: none"> 1.1. settaggio del dell'oggetto di tipo domainModel 1.2. Settaggio pesi delle matrici di co-occorrenza (CC, CQ, QC, QQ) 1.3. settaggio dell'oggetto ti tipo applicationAbstraction (chiama in automatico il metodo createGraph che genera il grafo)
Extensions:	<p>I settaggi si possono riunire in un unica chiamata al costruttore dell'oggetto di tipo GraphManager, passandogli tutti i parametri una volta sola) sia con pesi che senza.</p> <p>Possibilità di istanziare l'oggetto di tipo graphManager passando per costruttore direttamente un grafo.</p> <p>In caso di errori nelle matrici di co-occorrenza nell'oggetto di tipo ApplicationAbstraction, il grafo non verrà istanziato.</p>

ID:	UC3
Title:	Create SimplifyGraph
Description:	L'utente vuole creare un grafo semplificato, che rappresenta l'applicazione a microservizi finale.
Primary Actor:	User
Preconditions:	L'utente deve già aver istanziato un oggetto di tipo GraphManager con grafo interno istanziato.
Postconditions:	nell'attributo di GraphManager simplifiedGraph verrà istanziato il grafo semplificato e verrà ritornato il valore di loss
Main Success Scenario:	<ol style="list-style-type: none"> 1. Settaggio strategia di semplificazione: SimplifyGraphStrategy 2. Settaggio lossFunction: scelta oggetto concreto con interfaccia LossFunctionStrategy 3. esecuzione funzione simplifyAndComputeLoss (l'attributo simplifiedGraph sarà adesso istanziato) 4. chiamata a getSimplifiedGraph per ottenere il grafo semplificato
Extensions:	<p>Alternativamente si può eseguire i seguenti passaggi:</p> <ol style="list-style-type: none"> 3. Chiamata a funzione myBestSolution <p>Oppure fare una ricerca della migliore soluzione attraverso tutte le strategie disponibili:</p> <ol style="list-style-type: none"> 3. Chiamata a funzione findBestSolution <p>Una volta ottenuto il grafo semplificato, possiamo in più eseguire il seguente passaggio:</p> <ol style="list-style-type: none"> 5. Calcolo lossFunction dall'oggetto di tipo graphManager <p>In caso il grafo l'oggetto di tipo Graph non è istanziato, non verrà iniziata alcuna procedura di semplificazione.</p>

Soluzione Proposta

L'applicazione è divisa in due step:

1. Rappresentazione del Domain Model dell'applicazione monolitica come un grafo pesato.
2. Semplificazione del grafo tramite tagli in modo da generare i diversi microservizi.

In particolare:

INPUT

Il programma riceve in Input una sorta di “descrizione” dell'applicazione Monolitica e le indicazioni per il corretto svolgimento, ovvero:

- **Entità** del Domain model.
- **Accoppiamento** tra le varie entità.
- Gli **End Point** dell'applicazione monolitica descrivendo le entità che ne fanno parte.
- **Use case** dell'applicazione monolitica descrivendo gli End Point che ne fanno parte.
- **Strategie** da settare per lo svolgimento dell'applicazione nel modo desiderato. In particolare, l'utente dovrà scegliere preventivamente il metodo di costruzione delle matrici di co-occorrenza, la strategia di semplificazione del grafo e il modo per valutare le diverse soluzioni generate dall'applicazione (vedi cap x).
- **Peso** delle matrici di co-occorrenza (vedi cap. successivo)

SVOLGIMENTO

Modellazione del domain model come un grafo pesato

Partendo dalle entità e dagli accoppiamenti, si generano delle matrici di co-occorrenza, che racchiudono gli accoppiamenti di tutte le entità che fanno parte di un End Point.

Tra due entità A e B si hanno 4 diversi tipi di coupling, che in realtà diventano 8 per via della non-simmetria delle Matrici di co-occorrenza, in quanto queste sono delle matrici quadrate che possiedono come righe e come colonne le stesse entità nello stesso ordine. Ogni casella di questa matrice corrisponde ad una probabilità, che varia in base al tipo di matrice considerata (CC, CQ, QC, QQ) e alla posizione della casella, in particolare:

Considerando la matrice CC (Command-Command), e leggendo il valore presente in riga A e colonna B si trova:

$$P\{\text{Command on B} \mid \text{Command on A}\}$$

Si noti subito il motivo per cui queste matrici non sono simmetriche, dato che se leggessimo la stessa matrice in riga B colonna A si otterrebbe:

$$P\{\text{Command on A} \mid \text{Command on B}\}$$

Un'altra caratteristica interessante è che queste matrici non possiedono valori sulla diagonale, cioè non ha senso descrivere l'accoppiamento tra le entità A-A, B-B, C-C ecc..

Analogo accade con le altre 3 tipologie di matrici (CQ, QC, QQ), ad esempio, scegliendo CQ (Command-Query) si trova in linea A e colonna B:

$$P\{\text{Query on B} \mid \text{Command on A}\}$$

E viceversa in linea B colonna A:

$$P\{\text{Query on A} \mid \text{Command on B}\}$$

Dove per Command si intende "Lettura-da" e per Query si intende "Scrittura-su" le tabelle in cui sono mappate le Entities.

Ad ogni EndPoint saranno associate 4 matrici di co-occorrenza.

Ogni Use Case di un'applicazione monolitica, è formato da uno o più End Point, perciò, per calcolare le matrici di co-occorrenza a questo livello, occorre unire più matrici dello stesso tipo degli EndPoint che costituiscono tale Use Case. Queste matrici possono non avere le stesse entità nelle righe e nelle colonne. Stesso ragionamento vale quindi a livello di Business Logic essendo formato da più Use cases. Alla fine si arriverà ad avere 4 matrici di co-occorrenza che rappresentano più o meno l'intera applicazione monolitica.

Siccome potrebbero esserci più tecniche per creare delle matrici di co-occorrenza da altre già esistenti, è stato aggiunto il Design Pattern **Strategy** in modo da fornire la possibilità di inserire metodi nuovi e migliori senza dover cambiare tutto il codice.

Il passo successivo è quello di creare un grafo indiretto. Il grafo può essere creato a partire da qualsiasi livello: End Point, Use Case e Business Logic, ma ai fini del nostro obiettivo di creare diversi microservizi che descrivono l'applicazione monolitica di partenza, si farà riferimento al livello Business Logic.

Il grafo ha come nodi le entità e come archi gli accoppiamenti tra le varie entità. I pesi degli archi verranno calcolati con la formula seguente, utilizzando le 4 matrici di co-occorrenza:

$$E_{rc} = \frac{\sum_{i,j \in I} w_{ij} (M_{ij}[r][c] + M_{ij}[c][r])}{2w_{cc} + 2w_{cq} + 2w_{qc} + 2w_{qq}}$$

dove E_{rc} è un arco indiretto tra vertice **r** e **c**, quindi $E_{rc} = E_{cr}$.

$I = \{C, Q\}$, cioè i vari tipi di co-occorrenza, e $w_{cc}, w_{cq}, w_{qc}, w_{qq}$ sono 4 iperparametri presi in input che pesano i diversi tipi di co-occorrenza.

Con questo calcolo si riesce a generare un arco indiretto (dato che si prendono i valori della matrice in modo simmetrico), calcolato su ognuna delle 4 matrici, risparmiando così ulteriori operazioni.

Semplificazione del grafo

Ottenuto il grafo, lo step successivo è quello di tagliare gli archi del grafo in modo da avere le entità più disaccoppiate in microservizi diversi, mentre le entità più accoppiate in uno stesso microservizio. Per fare ciò si effettuano dei tagli degli archi del grafo cercando di dividerlo in più componenti connesse. L'algoritmo sviluppato per cercare di eseguire questa operazione è lo stesso descritto in ([Mazlami et al.](#)):

```

1: function CLUSTER(edges, npart, s)
2: for e  $\in$  edges do
3:   e.weight  $\leftarrow$  e.weight
4: end for
5: edgesMST  $\leftarrow$  KRUSKAL(edges)
6: edgesMST  $\leftarrow$  SORT(edgesMST )
7: edgesMST  $\leftarrow$  REVERSE(edgesMST )
8: n  $\leftarrow$  1
9: while n  $\leq$  npart do
10:  edgesMST [0].delete()
11:  n  $\leftarrow$  D FS(edgesMST )
12: end while
13: components  $\leftarrow$  REDUCECLUSTERS(edgesMST , s)
14: return components
15: end function
```

L'algoritmo cerca di dividere il grafo in ingresso in $npart$ componenti connesse e, in caso una di queste abbia un numero di nodi più grande di s , si cerca di dividere ulteriormente le componenti attraverso la funzione *REDUCECLUSTERS*. Inoltre è stato presentato un algoritmo di ricerca locale ispirato al simulated annealing che, iterando su questi due iperparametri, cerca la soluzione che dia il punteggio migliore, calcolato su delle **funzioni di loss**.

Le funzioni di Loss sono state utilizzate per dare un punteggio alle soluzioni che si calcolano con l'algoritmo di semplificazione. Ne sono state sviluppate 3:

- **MSE**
- **Reverse Huber**
- **EdgeLoss**, una tecnica ideata che tiene conto non solo degli archi tagliati ma anche di quelli che sono rimasti nella soluzione. (maggiori spiegazioni in [Progettazione](#))

I metodi di semplificazione del grafo, come quello appena accennato, possono essere molti, perciò abbiamo inserito il Design Pattern **Strategy**. Così facendo, possiamo comparare diverse soluzioni proposte da più algoritmi.

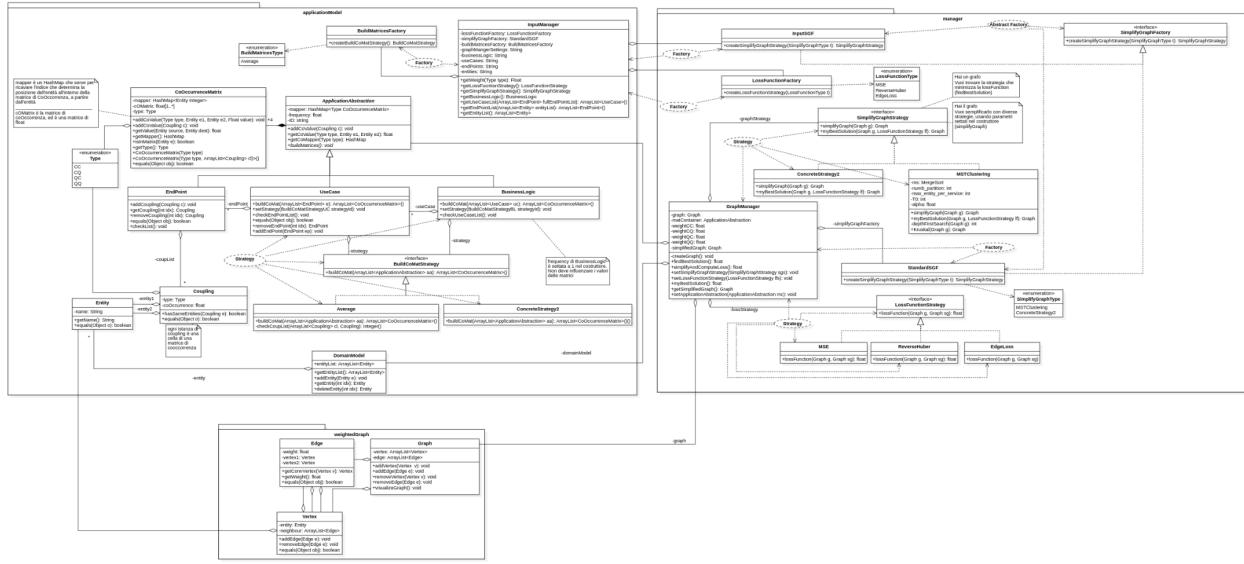
Quindi, riassumendo:

1. Partendo dalle entità e dagli accoppiamenti presi in Input, si generano delle matrici di co-occorrenza, che racchiudono gli accoppiamenti di tutte le entità che fanno parte di un End Point.
2. Si calcolano le matrici di Co-Occorrenza ai vari livelli (EndPoint, UseCase e BusinessLogic) utilizzando la strategia di costruzione delle matrici desiderata tra quelle presenti all'interno dello Strategy BuildCoMatStrategy.
3. Si trasformano le 4 matrici di co-occorrenza di livello Business Logic in un grafo pesato dove il peso dà conto dell'accoppiamento tra due entità.
4. Si semplifica il grafo (scegliendo tra diverse strategie).
5. Si compara le semplificazioni ottenute utilizzando una funzione di Loss.
6. Il risultato della semplificazione sarà un grafo costituito da più componenti connesse che rappresenteranno i diversi microservizi.

Progettazione della soluzione

Class Diagram

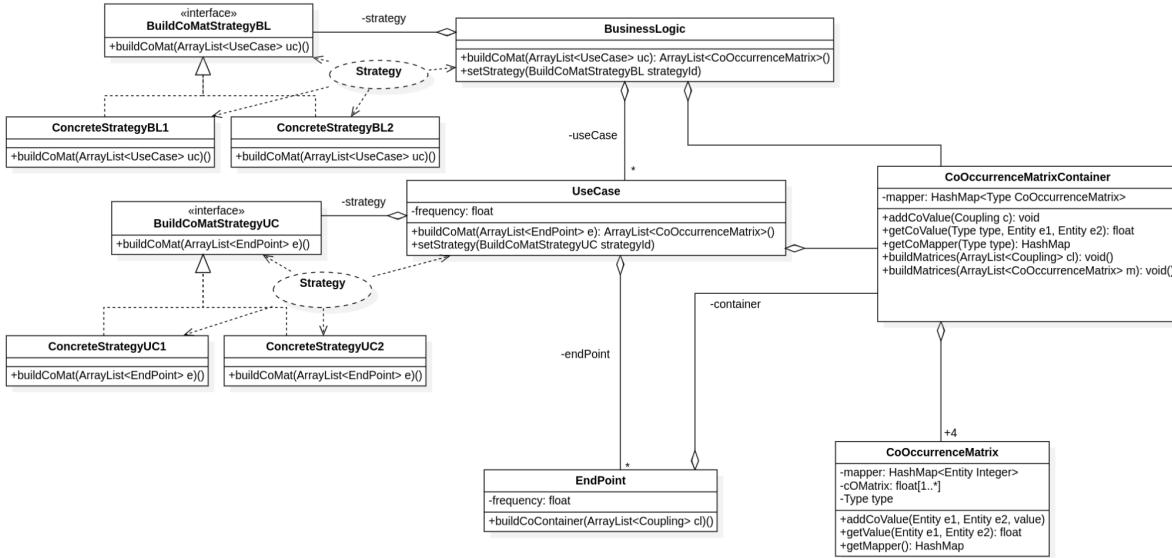
Di seguito proponiamo il Class Diagram del programma sviluppato:



Ovviamente nel corso del tempo ha subito svariate modifiche. Nelle sezioni successive le analizzeremo in dettaglio mostrando perché alla fine è stata scelta questa soluzione finale.

CoOccurrenceMatrix

Inizialmente questa classe era collegata ad un'altra classe chiamata *CoOccurrenceMatrixContainer*, la quale possedeva un hashmap per mappare le 4 matrici di co-occorrenza, utilizzando una enum class (Type) per indicare il tipo di co-occorrenza al fine di facilitarne l'accesso. Di conseguenza aveva 4 riferimenti alla classe *CoOccurrenceMatrix* e forniva dei metodi di accesso alla hashmap. Le classi *EndPoint*, *UseCase* e *BusinessLogic* possedevano un riferimento a *CoOccurrenceMatrixContainer* e tramite esso accedevano alle matrici, sfruttando i metodi di accesso che questa classe forniva.

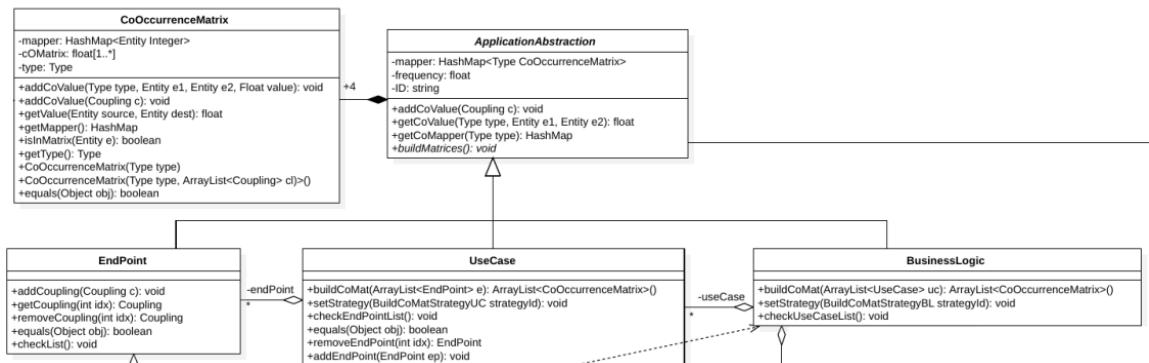


ApplicationAbstraction

EndPoint, *UseCase* e *BusinessLogic* avevano tutti necessità di accedere alle matrici e di invocare i metodi di accesso (che erano comuni a tutti). Per fare questo, avevano bisogno, ogni volta, di passare attraverso il loro riferimento a *CoOccurrenceMatrixContainer*.

Come si può notare dall'immagine sopra, *UseCase* che *BusinessLogic* avevano ognuno dei due uno strategy pattern individuale per decidere il metodo per generare le matrici di co-occorrenza. I due strategy pattern proponevano codice ridondante poiché alla fine si poteva tranquillamente eseguire le stesse procedure per creare matrici di co-occorrenza a partire da altre pur essendo su un livello di applicazione diverso (*UseCase* o *BusinessLogic*).

Per questo motivo, dopo uno studio più accurato, è stata proposta una soluzione più elegante e migliore. In particolare, è stato deciso di trasformare la classe *CoOccurrenceMatrixContainer* in *ApplicationAbstraction*, dove quest'ultima è una classe astratta, che viene estesa da *EndPoint*, *UseCase* e *BusinessLogic*.



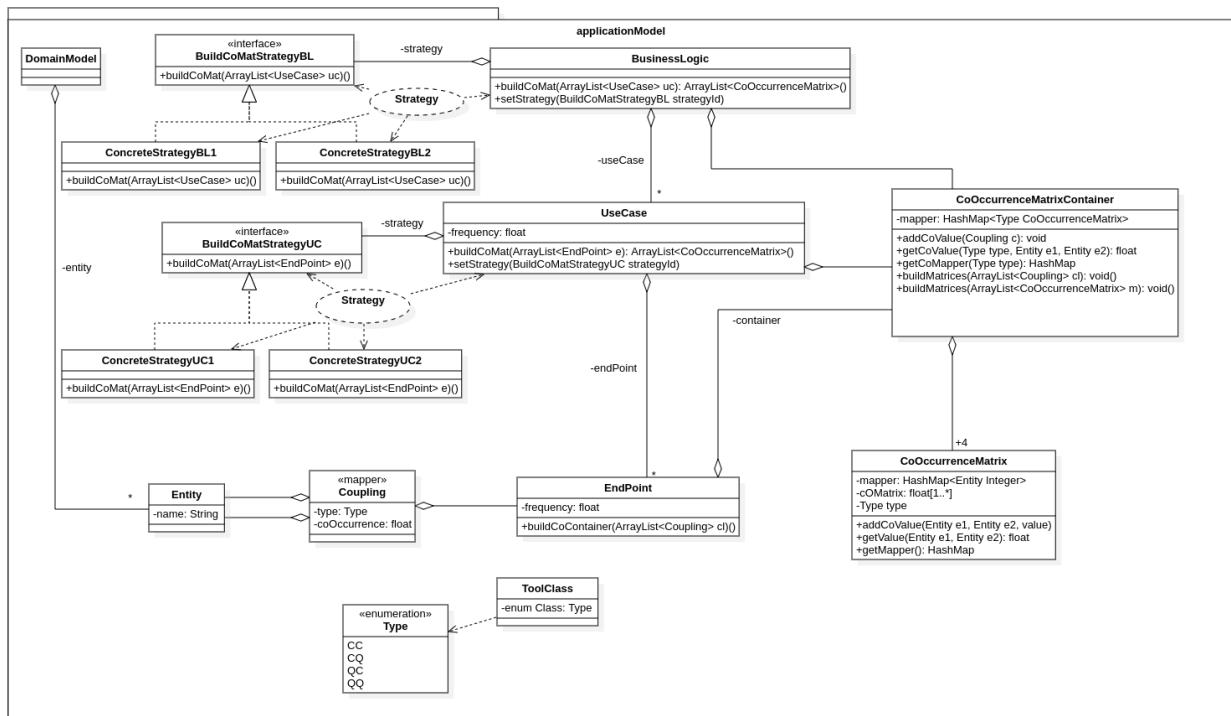
I vantaggi di questa soluzione sono i seguenti:

- È sufficiente un unico **strategy pattern**: questo prende come parametro un oggetto di tipo *ApplicationAbstraction*, riducendo in questo modo il codice che altrimenti sarebbe stato duplice.
- In questo modo poi si può ridurre la semplificazione da tutta l'applicazione monolitica fino a livello di *EndPoint*, generando così più casi d'uso.
- Adesso tutti e 3 (*EndPoint*, *UseCase*, *BusinessLogic*) possono accedere agli attributi che prima appartenevano a *CoOccurrenceMatrixContainer* e possono invocare i metodi di accesso alle matrici in modo diretto, senza passare dal riferimento.

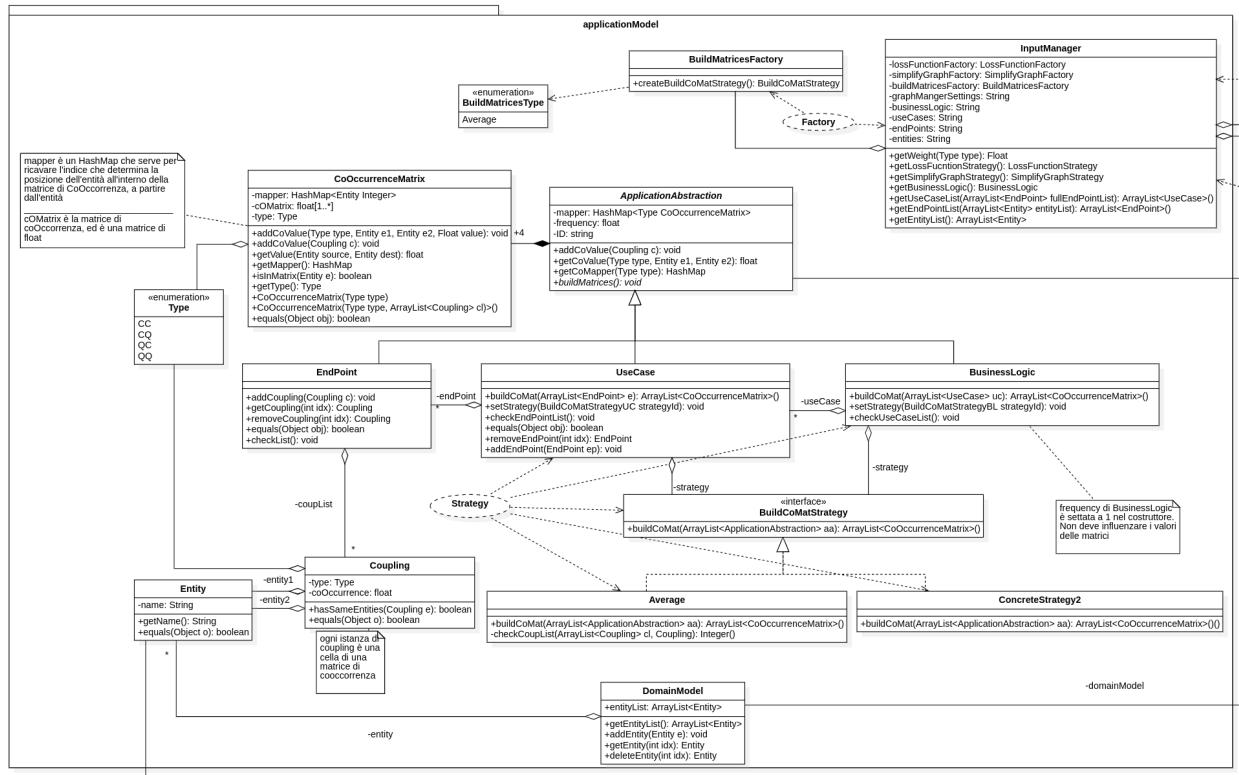
Altre modifiche apportate:

- L'attributo *frequency* presente in *EndPoint* e *UseCase* è stato inserito all'interno di *ApplicationAbstraction*, ma dato che *BusinessLogic* non aveva questo attributo, gli è stato assegnato come valore di default 1.
- È stata realizzata una sorta di “fake implementation” di *buildMatrices* in *EndPoint* dove, a differenza di quanto avviene in *UseCase* e *BusinessLogic*, non si richiama la funzione dello **Strategy**, ma bensì si creano le matrici di co-occorrenza da zero usando una lista di oggetti di tipo *Coupling*.

Di seguito mostriamo la vecchia versione del Package *ApplicationModel*:



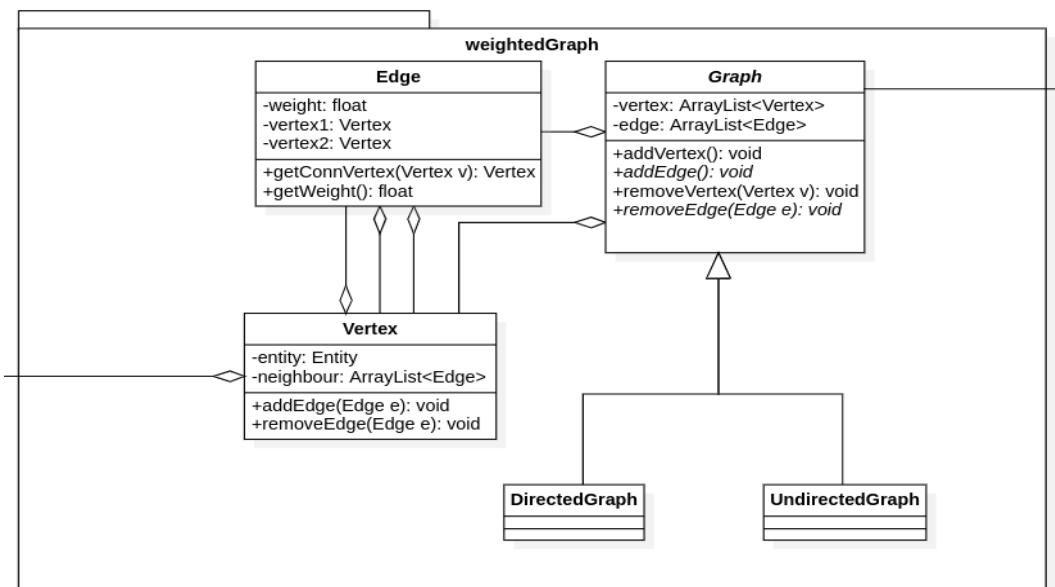
E infine la versione finale:



(dell'input ne parleremo successivamente)

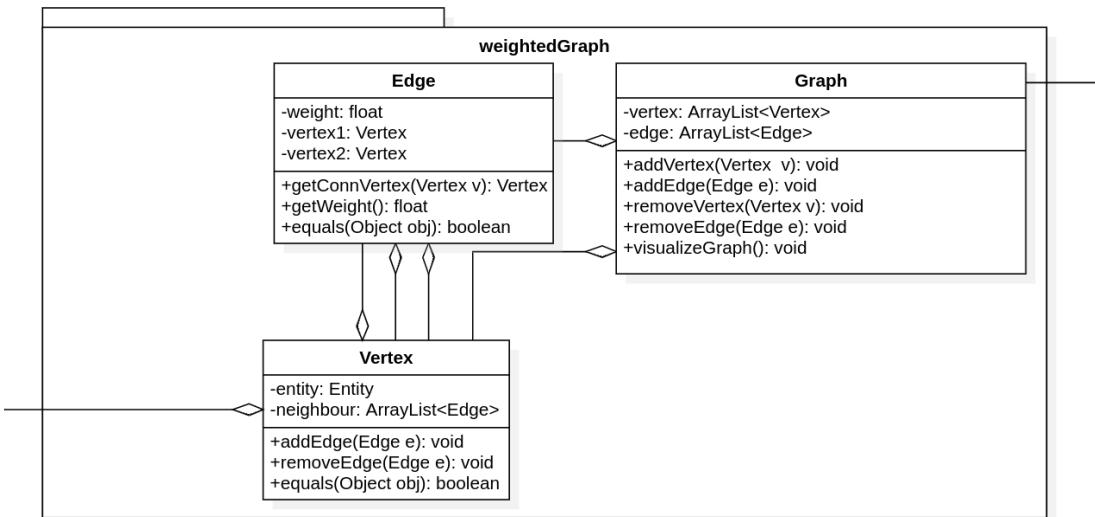
Weighted Graph

In questo progetto volevamo una struttura del grafo che prevedeva l'uso di un grafo diretto e indiretto:



Questa modalità di utilizzo del grafo, era stata inizialmente progettata per favorire la creazione di un grafo dalle 4 matrici di co-occorrenza: siccome le matrici sono asimmetriche, rappresentare prima un grafo diretto per poi trasformarlo in indiretto successivamente sembrava la soluzione migliore.

Dopo un'attenta analisi, siamo giunti alla conclusione che la costruzione del grafo indiretto poteva fin da subito essere fatta direttamente risparmiando così la costruzione di un intero grafo in più che differiva solo nell'immagazzinamento degli archi nei vertici. La versione finale di questa classe, risulta essere questa:



La struttura delle classi *Vertex* e *Edge* sono rimaste identiche:

- *Vertex*: la classe mantiene la lista dei suoi archi per favorire l'accesso ai vicini.
- *Edge*: mantiene i due vertici che l'arco collega e inoltre permette di ottenere il vertice direttamente connesso al vertice che viene passato in input, tramite il metodo `getConnVertex`.

La classe *Graph* mantiene due `ArrayList` di *Vertex* e *Edge* per facilitarne l'accesso nel grafo. Durante l'inserimento di un arco, la classe *graph* prevede di mantenere aggiornata la lista *neighbour*, inserendo il riferimento al nuovo arco (stesso comportamento accade in caso di taglio).

Design Patterns

All'interno del progetto abbiamo fatto uso dei seguenti Design Pattern:

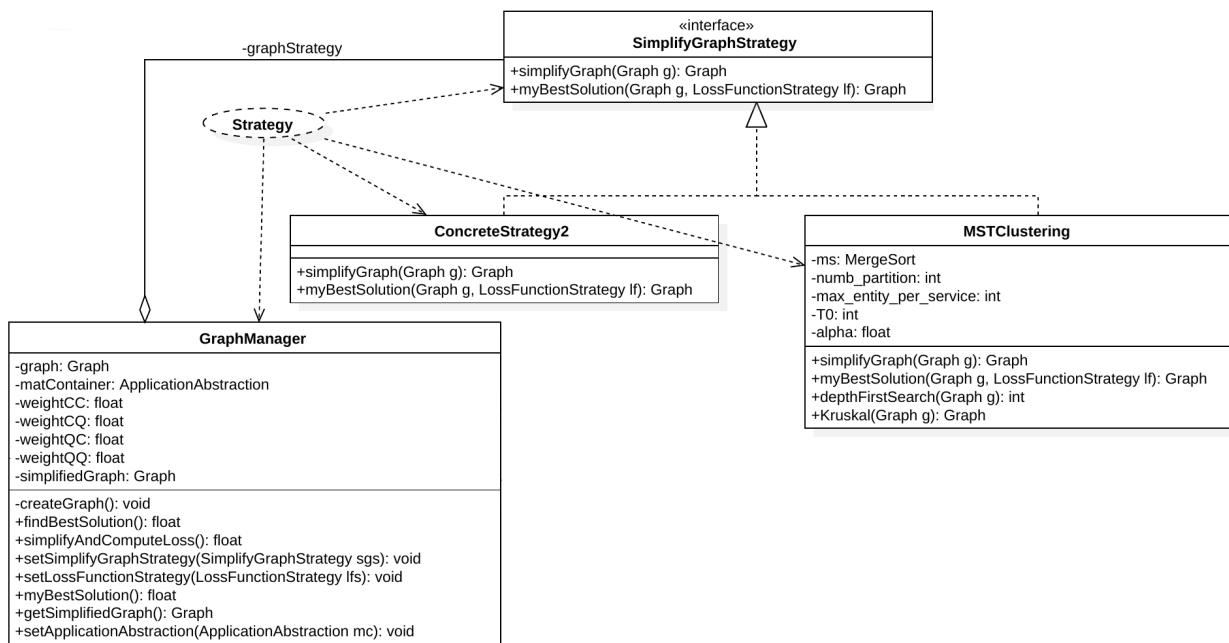
- **Strategy**
- **Factory**

STRATEGY

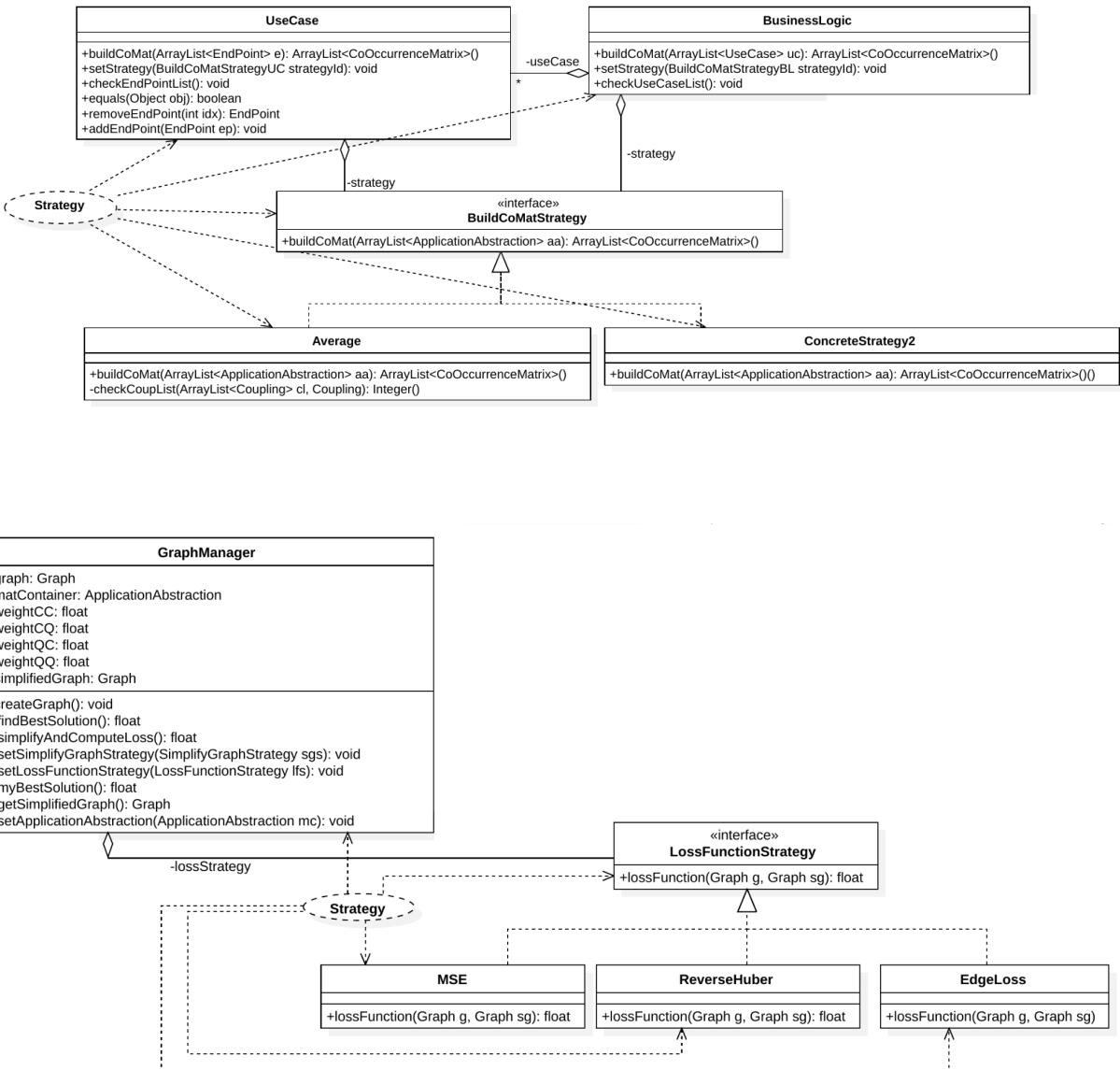
Lo **Strategy** è un Design Pattern comportamentale che permette di cambiare il comportamento di una classe a runtime, cambiando l'algoritmo utilizzato per svolgere una determinata mansione.

Abbiamo scelto di utilizzare il Design Pattern Strategy al fine di offrire all'utente la possibilità di effettuare numerose scelte, in particolare sugli algoritmi di semplificazione del grafo, di valutazione della loss e di costruzione delle matrici di co-occorrenza. In questo modo viene anche offerta all'utente la possibilità di effettuare delle valutazioni sui risultati provenienti dall'utilizzo di particolari combinazioni di algoritmi.

Di seguito mostriamo in dettaglio dove questo pattern viene utilizzato:



(la classe *ConcreteStrategy2* è stata mantenuta solo nel diagramma UML solo a scopo dimostrativo di come si può aggiungere con molta facilità altre classi che implementano lo stesso metodo ma in modo diverso).



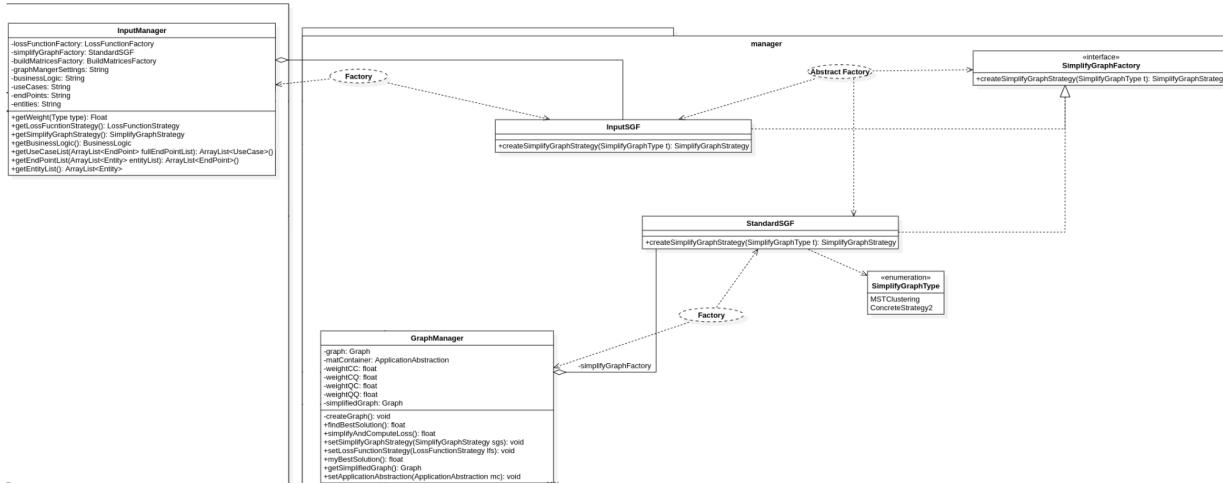
FACTORY

Il **Factory** è un Design Pattern creazionale che permette di istanziare oggetti più facilmente, senza dover specificare i dettagli.

Abbiamo deciso di utilizzare il Design Pattern Factory al fine di rendere più elegante, ordinata ed agevole la creazione automatica delle differenti strategie di semplificazione del grafo, di costruzione delle matrici di co-occorrenza e di costruzione delle funzioni di loss. In particolare viene utilizzato:

- all'interno del metodo "*findBestSolution()*" della classe *GraphManager*, in quanto questo metodo necessita di utilizzare tutte le diverse strategie di semplificazione del grafo, al fine di trovare la strategia che minimizza il valore della loss.
- all'interno della classe *InputManager*, per ottimizzare la creazione delle strategie di costruzione delle matrici di co-occorrenza, di semplificazione del grafo e di calcolo della loss, a partire dall'input dei file JSON.

Di seguito le parti del progetto dove lo abbiamo implementato:

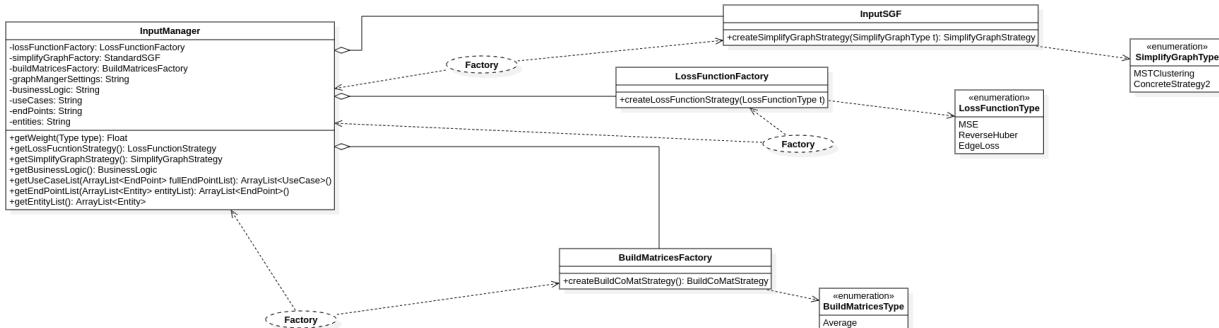


Recentemente è stato aggiunto un **abstract factory pattern** per lo strategy pattern *simplifyGraphStrategy*. Si definisce così l'interfaccia per generare diverse factory, in particolare:

- **StandardSGF**: che fornisce un metodo di creazione dell'oggetto concreto della strategia con parametri di default. Utilizzato all'interno del metodo *findBestSolution* della classe *GraphManager*.
- **InputSGF**: fornisce un metodo di creazione della strategia concreta prendendoli in input direttamente dall'utente.

Tenere la gestione dell'input all'interno di un metodo di creazione, genera un forte accoppiamento, però in questo modo si riesce a prendere dall'utente tutti i vari parametri delle diverse strategie in input indipendentemente da quali siano (e qui si risolve il solito problema: le strategie possono avere diversi parametri).

Di seguito le immagini degli altri **factory pattern**:

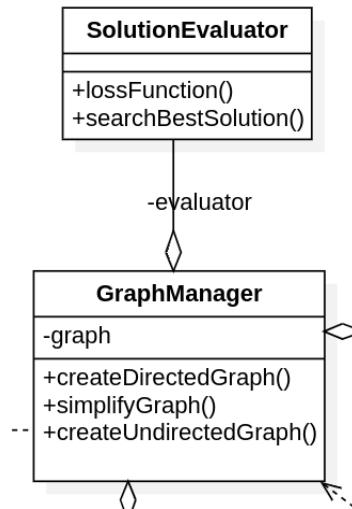


Come si può vedere, tutti questi **factory pattern** sono accompagnati da un'enum class:

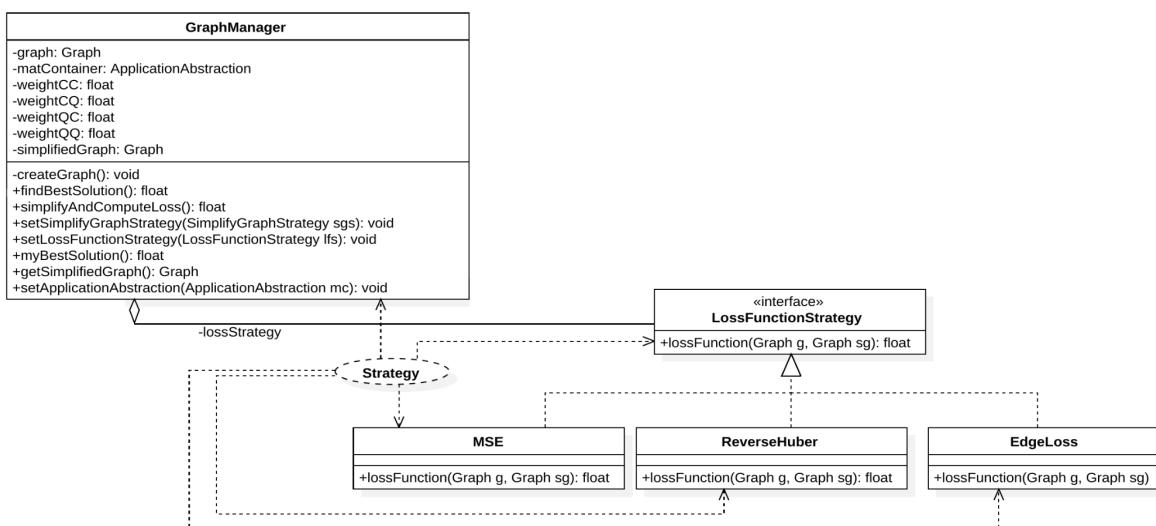
- *SimplifyGraphType*: definisce i vari tipi di strategie di semplificazione del grafo.
- *LossFunctionType*: definisce i vari tipi di funzione di loss.
- *BuildMatricesType*: definisce i tipi di strategia per la creazione di matrici (*BuildCoMatStrategy*).

Loss Functions

Inizialmente si prevedeva l'uso di un'unica funzione di loss di valutazione delle semplificazioni del grafo.



Attualmente è stato implementato uno strategy pattern che permette sia di usare diverse funzioni di loss da noi implementate, sia implementarne di nuove da zero rispettando l'interfaccia della strategia.



MSE:

- Errore quadratico medio sul peso degli archi tagliati: $\text{loss} = \sum_{e \in eL} (e)^2$, dove eL è l'insieme degli archi tagliati.

ReverseHuber:

- Permette di penalizzare maggiormente gli errori grandi rispetto a quelli piccoli,

$$\begin{cases} wt, & \text{se } wt < th. \\ \frac{wt^2 + th^2}{2th}, & \text{otherwise.} \end{cases}$$

utilizzando il quadrato del peso per gli errori grandi e il peso stesso per gli errori piccoli:

Dove:

- wt è il peso di un arco in percentuale(moltiplicato per 100)
- th : threshold, $th = 0.8(\max_{i \in eL}(wt_i))$. con eL sempre l'insieme degli archi tagliati.

EdgeLoss:

Siccome *MSE* e *ReverseHuber*, valutano la soluzione soltanto guardando gli archi tagliati, sicuramente porteranno a valutare una perdita minima quando l'applicazione risulta senza tagli. perciò è stata proposta una tecnica di loss da noi ideata che valuta la nuova soluzione in questo modo:

1. Tiene conto del numero di componenti connesse generate durante la tecnica di semplificazione, per fare ciò si effettua una stima del numero di componenti connesse desiderate per un determinato numero di entità:
 - $des = \text{stima}(x) = \frac{x}{\log_2 x}$, dove x è il numero di entità.
 - Si aggiunge poi alla loss:

$$\begin{cases} e^{2 * \frac{\text{numCC}}{des}}, & \text{se } \text{numCC} < des. \\ e^{2 * \frac{des}{\text{numCC}}}, & \text{otherwise.} \end{cases}$$

numCC è il numero di componenti connesse effettivo.

2. Dopodiché si calcola la media dei pesi degli archi in una componente connessa e se questa è inferiore ad una certa soglia, si aumenta il valore della loss in maniera più sostanziosa. In questo modo andiamo a premiare le soluzioni di microservizi con entità fortemente connesse:

$$\begin{cases} avg * 100, & \text{se } avg < \psi. \\ avg, & \text{otherwise.} \end{cases}$$

SimplifyGraphStrategy

Questo **Strategy pattern** prevedeva fin da subito di creare un metodo *simplifyGraph* che permette di semplificare il grafo con algoritmi diversi in modo intercambiabile. Il problema principale di questa soluzione era però che ogni algoritmo di semplificazione può avere iperparametri diversi (ad esempio per MSTClustering, abbiamo numero di partizione e max numero di entità per componente connessa). È stata quindi presa la decisione di eliminare gli iperparametri che si passano all'algoritmo di semplificazione, rendendoli attributi di classe concreta dello **strategy** permettendo di definirli al momento della costruzione dell'oggetto o usare direttamente quelli di default.

Inoltre è stato aggiunto il metodo *findBestSolution* nell'interfaccia dello **strategy**: La funzione è pensata per essere usata come algoritmo di ricerca locale dove, iterando sugli iperparametri dell'algoritmo e usando una funzione di loss passata per parametro, si cerca di trovare la soluzione migliore possibile.

In MSTClustering è stato proposto, come precedentemente detto, un metodo ispirato al simulated annealing:

```
1. function myBestSolution(lossFunctionStrategy lf)
2. for i ∈ (1, 1000) do:
3.   T =  $\frac{T_0}{i+\alpha}$ 
4.   last_n = this.numb_partition
5.   last_s = this.max_entity_per_service
6.   lastPath = choosePath(g, lastPath)
7.   result = simplifyGraph(g)
8.   Δ = lf.lossFunction(g, result)
9.   if Δ < 0
    a. best = lf.lossFunction(g, result)
    b. best_result = result
    c. best_s = max_entity_per_service
    d. best_n = numb_partition
10. else
    a. with p( $e^{-\frac{\Delta}{T}}$ ):
        i. numb_partition = last_n
        ii. max_entity_per_service = last_s
    b. otherwise:
        i. lastPath = 0
11. end for
12. numb_partition = best_n
13. max_entity_per_service = last_s
14. return best_result
```

La funzione *choosePath*, semplicemente prevede di selezionare casualmente una nuova combinazione dei parametri di *simplifyGraph*. *lastPath* è un intero che indica qual'è stata la vecchia strada intrapresa: questa indica se i parametri sono stati incrementati o diminuiti, entrambi o singolarmente. Quindi quando *lastPath* non viene azzerata, si continua ad andare in quella direzione, cioè ad esempio se *lastPath* indicava una strada dove si aumenta *numb_partition* e si diminuisce *max_entity_per_service*, allora *lastPath* non resettato continuerà a cambiare i parametri nella stessa direzione.

GraphManager

La classe prevede una gestione completa del grafo dalla sua creazione alla sua semplificazione e valutazione usando le tecniche di loss. Per questo gli **strategy pattern** precedentemente discussi quali *SimplifyGraphStrategy* e *LossFunctionStrategy* vengono utilizzati in questa classe. Inoltre è stato poi sviluppato un metodo chiamato *findBestSolution* che si occupa di istanziare tutte le strategie di semplificazione del grafo disponibili per poi iterativamente invocare su ognuna il proprio metodo *myBestSolution*. La funzione si salva infine il risultato della strategia che ha riscontrato il miglior punteggio. Questo metodo quindi permette di confrontare tutte le strategie di semplificazione rispetto ad una specifica funzione di loss passata come attributo di GraphManager.

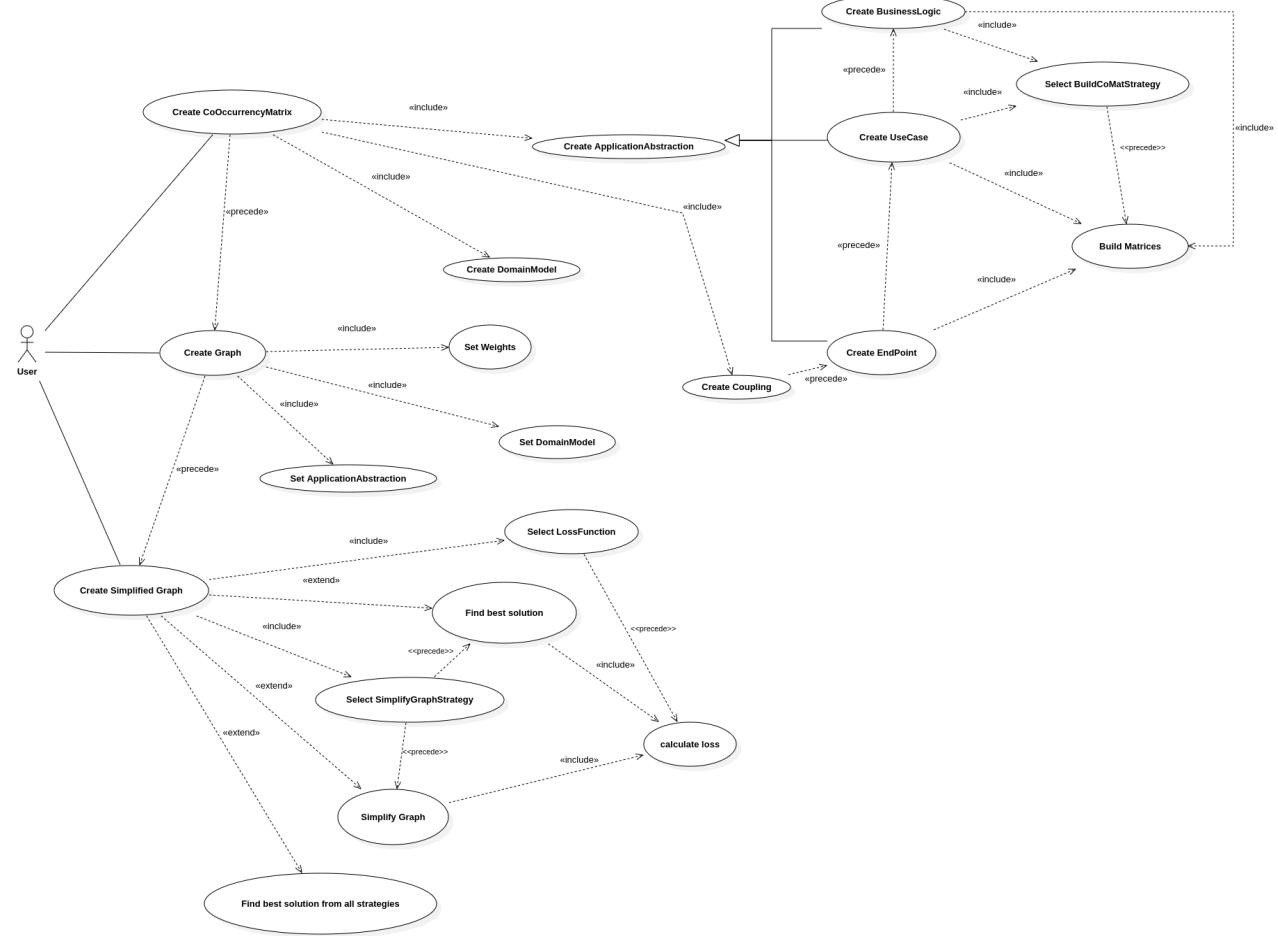
GraphManager
<pre>-graph: Graph -matContainer: ApplicationAbstraction -weightCC: float -weightCQ: float -weightQC: float -weightQQ: float -simplifiedGraph: Graph</pre>
<pre>-createGraph(): void +findBestSolution(): float +simplifyAndComputeLoss(): float +setSimplifyGraphStrategy(SimplifyGraphStrategy sgs): void +setLossFunctionStrategy(LossFunctionStrategy lfs): void +myBestSolution(): float +getSimplifiedGraph(): Graph +setApplicationAbstraction(ApplicationAbstraction mc): void</pre>

Un altro dettaglio molto particolare riguarda l'introduzione di *ApplicationAbstraction*: È stato deciso di cambiare il tipo dell'attributo di *matContainer* da *BusinessLogic* a *ApplicationAbstraction* per avere la possibilità di generare un grafo che rappresenti un End

Point o uno Use Case dell'applicazione monolitica, espandendo così i casi di utilizzo della nostra applicazione.

Aggiornamento Use Case Diagram

Questa scelta progettuale ha fatto sì di espandere i possibili casi d'uso in questo modo:



InputManager

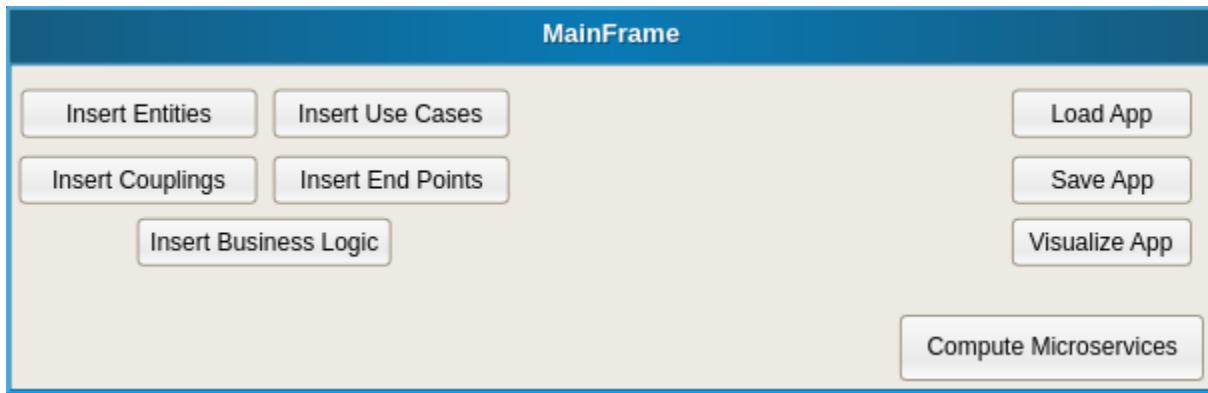
Per gestire l'input abbiamo pensato di sviluppare un metodo che usa dei file JSON per caricare tutte le informazioni dell'applicazione monolitica e le varie informazioni di configurazione dell'applicazione come quale strategia o quale Loss Function utilizzare. Questo metodo di input sarebbe poi utilizzabile per una futura espansione per salvare le varie configurazioni dell'input o delle soluzioni trovate dall'applicazione. Per fare ciò sono stati realizzati diversi **factory pattern** per facilitare la costruzione di questi metodi come già esposto precedentemente.

Inizialmente, per ottenere l'input, si pensava ad uno classico da tastiera. Questa soluzione è stata molto presto scartata dato che si sarebbe tradotta in un lavoro di scrittura immenso per l'utente (ogni singola componente doveva essere scritta tutte le volte che l'applicazione fosse stata avviata). Con il nostro metodo, invece, l'utente deve scrivere le componenti

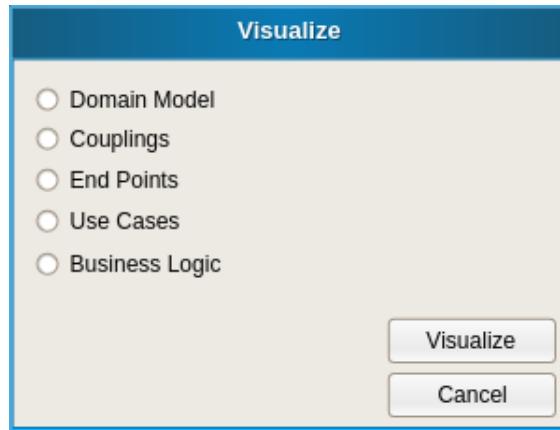
dell'applicazione una volta soltanto, permettendo così il riutilizzo e non solo: Scrivere tutte le componenti su una riga di comando sarebbe stato molto lento e prone agli errori, invece, con il metodo proposto, una volta scritte tutte le componenti, si riesce in modo molto semplice a navigare attraverso tutti i file e quindi scorgere in modo molto facile eventuali errori di scrittura.

Mockups

Di seguito vengono proposti dei mockups di una possibile implementazione di un front-end. Il software utilizzato è Pencil.



Qui possiamo vedere l'interfaccia principale dell'applicativo, permette di inserire i vari elementi di una applicazione, salvare l'applicazione caricata così da facilitarne il caricamento per usi futuri e visualizzare tutti gli oggetti dell'applicazione.



Nella fase di inserimento si prevede un ordine prestabilito: Entity, Couplings, EndPoints, UseCases e infine BusinessLogic. I bottoni saranno disattivati finché non sarà rispettato l'ordine di precedenza. Ognuno dei bottoni dell'interfaccia principale apre una nuova

finestra che permette di inserire uno o più elementi di quel tipo (a parte per Business Logic di cui se ne prevede una sola). Di seguito sono riportate le varie interfacce per gli input.

Insert Entities

Insert the name of the entity(unique):

Frequency:

Insert Couplings

Select the first Entity

Select the second Entity

Select the type

Value(0÷1):

Insert End Points

Name of the End Point(unique):

Frequency:

Insert Use Cases

Name of the Use Case(unique):

Frequency:

Select strategy for building matrices

	Entity 1	Entity 2	Type
<input checked="" type="checkbox"/>			CC
<input checked="" type="checkbox"/>			CQ
<input type="checkbox"/>			CC
<input checked="" type="checkbox"/>			QC
<input checked="" type="checkbox"/>			QQ

	End Point Name
<input checked="" type="checkbox"/>	
<input type="checkbox"/>	
<input type="checkbox"/>	

Insert Business Logic

Name of the Business Logic(unique):

Frequency: float

Select strategy for building matrices
▼

Insert
Cancel

Una volta eseguito tutto l'input (almeno fino ad EndPoint), verrà attivato il tasto **Compute Microservices** nella finestra *MainFrame* che permetterà di cominciare la procedura atta a semplificare l'applicazione monolitica in ingresso in una possibile applicazione a microservizi. Una volta premuto il tasto, comparirà la seguente finestra:

Select level

End Point
 Use Case
 Business Logic

Select ApplicationAbstraction ▼

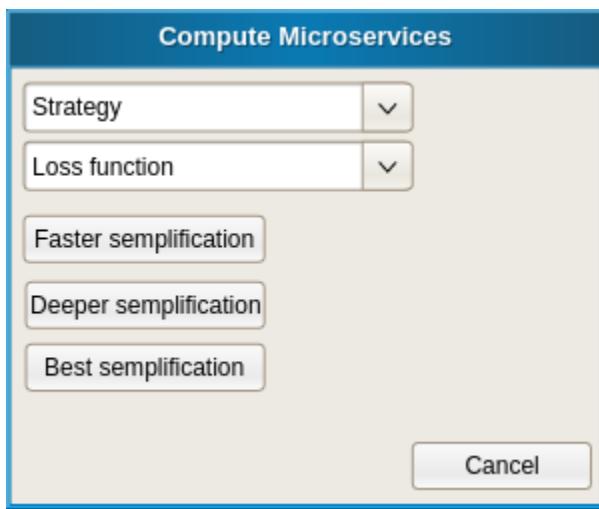
Insert Weight for:

CC:	decimal
CQ:	decimal
QC:	decimal
QQ:	decimal

Next
Cancel

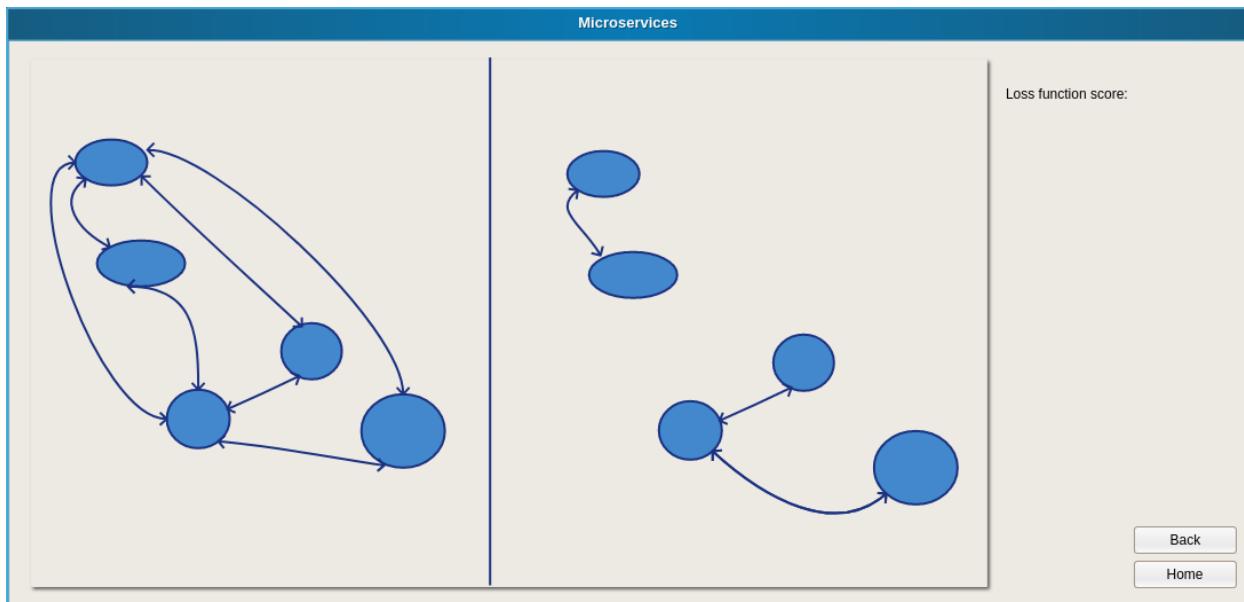
Questa è una finestra che permette di scegliere a quale livello (EndPoint, UseCase o BusinessLogic) eseguire l'algoritmo di semplificazione. In caso di **EndPoint** o **UseCase** verrà anche attivato il menù a tendina che permetterà di scegliere su quali, tra gli **EndPoint** o gli **UseCase**, eseguire l'algoritmo di semplificazione. Inoltre Vengono poi aggiunti dei pesi a descrizione dell'utente sui vari tipi di co-occorrenza (lasciarli vuoti significa = 1).

Una volta inserite tutte le informazioni, cliccando su next, si procede nella prossima finestra:



Qui si apre la scelta di quale loss function usare (se sono state implementate), oppure quale strategia di semplificazione usare. Infine, a discapito dell'utente, è possibile scegliere mediante i 3 tasti a sinistra, il tipo di semplificazione: I primi due tasti dipendono dalla scelta della strategia di semplificazione e dalla loss function perciò saranno attivati solo quando entrambe saranno state scelte. L'ultimo dipende solo dalla loss function perciò basta la scelta di quest'ultima per poter attivare il bottone **Best simplification**.

Una volta premuto una delle 3 scelte, si aprirà la seguente finestra:



Qui si può analizzare la soluzione data dall'algoritmo in forma di grafo: il prima (a sx) e il dopo (a dx) della trasformazione in microservizi, riportando sulla destra il punteggio della loss function.

Possibili aggiunte

Elenchiamo di seguito possibili implementazioni aggiuntive per questo Software:

- **Sviluppo di un front-end** con interfaccia grafica al fine di facilitare l'inserimento dei dati all'utente.
- Aggiungere la possibilità di **salvare i risultati sul disco**, in modo tale che l'utente possa caricare risultati precedentemente ottenuti e proseguire il suo lavoro dal punto in cui era rimasto.

Implementazione

Classi ed Interfacce

CoOccurrenceMatrix

Contiene i dati relativi alla co-occorrenza tra le entità di una data ApplicationAbstraction. I dati sono memorizzati in una matrice quadrata, dove le righe e le colonne sono associate alle entità contenute nella ApplicationAbstraction tramite una HashMap.

```
public class CoOccurrenceMatrix {  
  
    /**  
     * @param type the matrix's type of co-occurrence  
     */  
    public CoOccurrenceMatrix(Type type) {  
        mapper = new HashMap<Entity, Integer>();  
        this.type = type;  
    }  
    /**  
     * constructor that take all the coupling by list.  
     * @param type  
     * @param cl  
     */  
    public CoOccurrenceMatrix(Type type, ArrayList<Coupling> cl){  
        this(type);  
        int map_index = 0;  
        for(int i = 0; i < cl.size(); i++){  
            if(cl.get(i).getType() == this.type){ //final check for same type of coupling  
                if(!isInMatrix(cl.get(i).getSrcEntity()))  
                    mapper.put(cl.get(i).getSrcEntity(), map_index++);  
  
                if(!isInMatrix(cl.get(i).getDestEntity()))  
                    mapper.put(cl.get(i).getDestEntity(), map_index++);  
            }  
        }  
        this.cOMatrix = new float[mapper.size()][mapper.size()];  
        int x,y;  
        for(int i = 0; i < cl.size(); i++){  
            if(cl.get(i).getType() == this.type){  
                x = mapper.get(cl.get(i).getSrcEntity());  
                y = mapper.get(cl.get(i).getDestEntity());  
                cOMatrix[x][y] = cl.get(i).getCoOccurrence();  
            }  
        }  
    }  
}
```

Permette anche di verificare se un'entità è presente o meno nelle righe e colonne della matrice tramite il metodo isInMatrix(Entity e).

```

/**
 * function that check if an entity is already in the matrix,
 * @param e entity to check
 * @return true if the entity is in the matrix, otherwise false
 */
private Boolean isInMatrix(Entity e){
    Iterator<Entity> it = this.mapper.keySet().iterator();
    while(it.hasNext()){
        Entity x = it.next();
        if(e.equals(x)){
            return true;
        }
    }
    return false;
}

```

[ApplicationAbstraction, EndPoint, UseCase e BusinessLogic](#)

ApplicationAbstraction è una classe astratta che definisce attributi e metodi usati dalle classi EndPoint, UseCase e BusinessLogic. Contiene 4 matrici di cooccorrenza, associate al rispettivo tipo tramite una HashMap. Espone il metodo astratto buildMatrices() per fare in modo che ognuna delle classi derivate possa costruire le sue matrici di cooccorrenza in modo diverso. EndPoint costruisce le matrici di cooccorrenza tramite una lista di Coupling data in input, mentre UseCase e BusinessLogic le costruiscono utilizzando una BuildCoMatStrategy.

```

public abstract class ApplicationAbstraction {

    /**
     * Default constructor
     */
    public ApplicationAbstraction(String id, float f) {
        mapper = new HashMap<Type, CoOccurrenceMatrix>();
        this.ID = id;
        this.frequency = f;
    }
}

```

```
public class EndPoint extends ApplicationAbstraction {

    /**
     *
     * @param cl the arrayList of Coupling
     * @param ID
     * @param freq
     * automatically creates the matrices
     */
    public EndPoint(ArrayList<Coupling> cl, String ID, float freq){
        super(ID, freq);
        this.coupList = new ArrayList<Coupling>();
        for(int i = 0; i< cl.size(); i++)
            this.coupList.add(cl.get(i));
        buildMatrices();
    }

    /**
     *
     * @param ID
     * @param freq
     */
    public EndPoint(String ID, float freq){
        super(ID, freq);
        this.coupList = new ArrayList<Coupling>();
    }
}
```

```

public class UseCase extends ApplicationAbstraction {

    /**
     *
     * @param id the id of the use case
     * @param f the frequency of the use case
     * @param aa list of End Points
     *
     */
    public UseCase(String id, float f, ArrayList<EndPoint> epl){
        this(id,f);
        this.endPointList = new ArrayList<EndPoint>();
        for(int i = 0; i< epl.size(); i++)
            this.endPointList.add(epl.get(i));
    }

    /**
     *
     * @param id the id of the use case
     * @param f the frequency of the use case
     * @param aa list of End Points
     * @param strat the strategy to build the matrices
     * automatically creates the matrices
     */
    public UseCase(String id, float f, ArrayList<EndPoint> epl, BuildCoMatStrategy strat){
        this(id,f,epl);
        this.strategy = strat;
        buildMatrices();
    }
}

```

```

public class BusinessLogic extends ApplicationAbstraction {

    /**
     *
     * @param id the id of the Business Logic
     */
    public BusinessLogic(String id) {
        super(id, f: 1);
        useCaseList = new ArrayList<UseCase>();
    }

    /**
     *
     * @param id the id of the Business Logic
     * @param ucl the array
     */
    public BusinessLogic(String id, ArrayList<UseCase> ucl) {
        this(id);
        for(int i = 0; i < ucl.size(); i++)
            this.useCaseList.add(ucl.get(i));
    }

    /**
     *
     * @param id
     * @param ucl
     * @param strat
     * automatically build matrices
     */
    public BusinessLogic(String id, ArrayList<UseCase> ucl, BuildCoMatStrategy strat) {
        this(id, ucl);
        this.strategy = strat;
        buildMatrices();
    }
}

```

Coupling

È una classe che contiene un singolo valore di cooccorrenza tra due entità. I dati di input verranno memorizzati in istanze di questa classe, che andranno a loro volta a formare le matrici di cooccorrenza degli EndPoint. Ogni istanza di Coupling corrisponderà a una singola casella di una matrice di cooccorrenza.

```

public class Coupling {

    /**
     * @param e1 the first entity in the coupling (is directional)
     * @param e2 the second entity in the coupling
     * @param type param that show what type of Coupling is, selected from 4 different enum: CC,CQ,QC,QQ
     * @param coc the actual value of the coupling
     */
    public Coupling(Entity e1, Entity e2, Type type, float coc) {
        this.type = type;
        this.coOccurrence = coc;
        this.entity1 = e1;
        this.entity2 = e2;
    }
}

```

DomainModel

Questa classe rappresenta il domain model dell'applicazione monolitica restful di partenza. In particolare, contiene una lista di tutte le entità dell'applicazione di partenza.

```

public class DomainModel {

    /**
     * @param es Array of entity
     */
    public DomainModel(Entity[] es) {
        entityList = new ArrayList<>();
        for(int i=0;i<es.length;i++){
            entityList.add(es[i]);
        }
    }
}

```

BuildCoMatStrategy e BuildMatricesFactory

BuildCoMatStrategy è l'interfaccia dello strategy pattern utilizzato per creare le matrici di cooccorrenza di UseCase e BusinessLogic. Le strategie concrete devono implementare il metodo buildCoMat(), che crea le matrici di cooccorrenza degli UseCase e della BusinessLogic, unendo le matrici di cooccorrenza delle ApplicationAbstraction di livello inferiore che essi contengono (degli EndPoint nel caso di uno UseCase, degli UseCase nel caso della BusinessLogic).

```

public class BuildMatricesFactory {

    /**
     * Default constructor
     */
    public BuildMatricesFactory() {
    }

    /**
     * @param BuilMatricesType t name of the concrete strategy to be instantiated
     * @return reference to instance of concrete strategy implementing the chosen strategy
     */
    public BuildCoMatStrategy createBuildCoMatStrategy(BuildMatricesType t) {

        switch (t) {
            case Average:
                return new Average();

            default:
                return null;
        }
    }
}

```

BuildMatricesFactory ha il compito di istanziare dinamicamente la strategia scelta per la creazione delle matrici di cooccorrenza.

GraphManager

Questa classe ha lo scopo di costruire e contenere il grafo generato dalle matrici di cooccorrenza di una ApplicationAbstraction, e di invocare una SimplifyGraphStrategy e una LossFunctionStrategy, al fine di scomporre il grafo tagliando determinati archi, minimizzando la funzione di loss.

```

public class GraphManager {

    /**
     * Constructor that receives all attributes
     */
    public GraphManager(ApplicationAbstraction mc, float wCC, float wCQ, float wQC, float wQQ, SimplifyGraphStrategy sgs, LossFunctionStrategy lfs, DomainModel dm) {
        this.domainModel = dm;
        this.matContainer = mc;
        this.weightCC = wCC;
        this.weightCQ = wCQ;
        this.weightQC = wQC;
        this.weightQQ = wQQ;
        this.graphStrategy = sgs;
        this.lossStrategy = lfs;
        this.graph = new Graph();
        if(!this.createGraph()){
            System.out.println("Error in creating graph");
        }
        this.simplifyGraphFactory = new SimplifyGraphFactory();
        this.simplifiedGraph = null;
    }
}

```

La funzione `simplifyAndComputeLoss()` permette di semplificare il grafo con la strategia contenuta in `GraphManager` con dei parametri di default.

```

    /**
     * Simplify the graph using the SimplifyGraphStrategy set and save the result in simplifiedGraph attribute
     * @return return the loss value of the simplification made
     */
    public Float simplifyAndComputeLoss() {
        if (this.graphStrategy == null)
            return null;
        if (this.graph == null)
            return null;
        //INVOCARE simplifyGraph usando la graphStrategy preimpostata
        this.simplifiedGraph = this.graphStrategy.simplifyGraph(this.graph); //RETURN il grafo restituito dalla funzione simplifyGraph
        // INVOCARE lossFunction della Strategia di Loss settata e ritornare il suo valore
        return this.lossStrategy.lossFunction(this.graph, this.simplifiedGraph);
    }
}

```

La funzione `myBestSolution()` calcola la semplificazione migliore con la strategia contenuta in `GraphManager`, iterando sui parametri specifici della strategia.

```

    /**
     * Calculate the best graph cut for a specific SimplifyGraphStrategy set, varying the internal parameters of the Strategy
     * and save the best graph in the attribute simplifiedGraph
     * @return the loss value for the best graph cut. Save the best graph in the attribute simplifiedGraph
     */
    public Float myBestSolution() {
        if (this.graphStrategy == null)
            return null;
        if (this.graph == null)
            return null;
        this.simplifiedGraph = this.graphStrategy.myBestSolution(this.graph, this.lossStrategy);
        return this.lossStrategy.lossFunction(this.graph, this.simplifiedGraph);
    }
}

```

La funzione `findBestSolution()` permette di iterare su tutte le strategie di semplificazione implementate, calcolando la soluzione migliore per ciascuna strategia, confrontandole e scegliendo la migliore.

```
public Float findBestSolution() {  
    if (this.graphStrategy == null)  
        return null;  
  
    if (this.graph == null)  
        return null;  
  
    //SETTARE il valore della lossFunction al massimo  
    float bestLossValue = Float.MAX_VALUE;  
    float currentLossValue;  
    Graph bestGraph = null;  
    Graph currentGraph = new Graph();  
    //ITERARE per ogni SimplifyGraphType  
    for (SimplifyGraphType t : SimplifyGraphType.values()) {  
  
        //GENERARE la SimplifyGraphStrategy con la SimplifyGraphFactory passando il SimplifyGraphType  
        SimplifyGraphStrategy s = this.simplifyGraphFactory.createSimplifyGraphStrategy(t);  
  
        //INVOCARE myBestSolution della Concretestrategy (che invocherà il suo simplifyGraph variando i suoi parametri)  
        currentGraph = s.myBestSolution(this.graph, this.lossStrategy);  
  
        //CALCOLARE il lossValue utilizzando la lossFunction  
        currentLossValue = this.lossStrategy.lossFunction(this.graph, currentGraph);  
  
        //CONFRONTARE il valore ottenuto con il lossValue attuale e sostituirlo in caso fosse migliore  
        if (Float.compare(currentLossValue, bestLossValue) < 0) {  
            bestGraph = currentGraph;  
            bestLossValue = currentLossValue;  
        }  
    }  
  
    this.simplifiedGraph = bestGraph;  
  
    return bestLossValue;  
}
```

Tutte e tre queste funzioni ritornano il valore di loss della semplificazione, e salvano il grafo semplificato dentro `GraphManager`.

[SimplifyGraphStrategy](#), [SimplifyGraphFactory](#), [LossFunctionStrategy](#) e [LossFunctionFactory](#)

`SimplifyGraphStrategy` e `LossFunctionStrategy` sono le interfacce rispettivamente dello strategy pattern utilizzato per semplificare il grafo e di quello utilizzato per stimare il costo in termini di IPC che una particolare semplificazione del grafo comporta. Le strategie di semplificazione concrete devono implementare un metodo `simplifyGraph()` per semplificare il grafo usando dei parametri di default, e un metodo `myBestSolution()` per iterare la semplificazione del grafo di partenza con diversi parametri, per trovare la semplificazione migliore.

```

public class SimplifyGraphFactory {

    /**
     * Default constructor
     */
    public SimplifyGraphFactory() {
    }

    /**
     * @param SimplifyGraphType t name of the concrete strategy to be instantiated
     * @return reference to instance of concrete strategy implementing the chosen strategy
     */
    public SimplifyGraphStrategy createSimplifyGraphStrategy(SimplifyGraphType t) {

        switch (t) {
            case MSTClustering:
                return new MSTClustering();

            default:
                return null;
        }
    }
}

```

SimplifyGraphFactory e LossFunctionFactory hanno rispettivamente il compito di istanziare dinamicamente la strategia di semplificazione e la funzione di loss scelta.

```

public class LossFunctionFactory {

    /**
     * Default constructor
     */
    public LossFunctionFactory() {
    }

    /**
     * @param LossFunctionType t name of the concrete strategy to be instantiated
     * @return reference to instance of concrete strategy implementing the chosen strategy
     */
    public LossFunctionStrategy createLossFunctionStrategy(LossFunctionType t) {

        switch (t) {
            case CutSum:
                return new ReverseHuber();

            case LossStrategy2:
                return new MSE();

            default:
                return null;
        }
    }
}

```

Graph, Edge e Vertex

Queste tre classi implementano la struttura del grafo. La classe Graph contiene una lista di tutti gli archi e una di tutti i vertici, insieme ai metodi per aggiungere e rimuovere elementi dalle liste.

```

public class Graph {

    public Graph() {
        vertexList = new ArrayList<Vertex>();
        edgeList = new ArrayList<Edge>();
    }

    public Graph(Graph g){
        this.vertexList = new ArrayList<Vertex>();
        this.edgeList = new ArrayList<Edge>();
        for(int i = 0; i < g.getVertexList().size(); i++){
            addVertex(new Vertex(g.getVertexList().get(i).getEntity()));
        }

        Vertex v2 = null;
        Vertex v1 = null;
        for(int i = 0; i < g.getEdgeList().size(); i++){
            for(Vertex v : vertexList){
                if(v.equals(g.getEdgeList().get(i).getVertex1()))
                    v1 = v;

                if(v.equals(g.getEdgeList().get(i).getVertex2()))
                    v2 = v;
            }
            if(v1 != null && v2 != null)
                addEdge(new Edge(g.getEdgeList().get(i).getWeight(), v1, v2));
        }
    }
}

```

La classe Edge rappresenta gli archi, contiene il peso dell'arco, i due vertici connessi, e un metodo getConnVertex(Vertex v) che ricevendo uno dei due vertici dell'arco ritorna l'altro. Gli archi sono adirezionali.

```
public class Edge {  
  
    public Edge(float weight, Vertex v1, Vertex v2) {  
        this.weight = weight;  
        this.vertex1 = v1;  
        this.vertex2 = v2;  
    }  
  
    /**  
     * @param Vertex v one of the vertices connected by the edge  
     * @return the vertex connected to v by the edge  
     */  
    public Vertex getConnVertex(Vertex v) {  
        if (v.equals(vertex1))  
            return vertex2;  
        if (v.equals(vertex2))  
            return vertex1;  
        return null;  
    }  
}
```

La classe Vertex rappresenta i vertici, contiene una lista di tutti gli archi a cui è connesso il vertice.

```
public class Vertex {  
  
    public Vertex(Entity e) {  
        entity = e;  
        neighbour = new ArrayList<Edge>();  
    }  
}
```

Unit Test

Siccome ogni classe è fortemente collegata alle altre, è stata presa la decisione di testare la maggior parte del codice per evitare il maggior numero di bugs. Nel progetto è stato usato il framework JUnit 4.13.2.

AppModelTest

In questa classe di test sono state testate tutte le classi appartenenti al package **ApplicationModel**. In particolare sono stati testati i metodi di creazione delle matrici, e selezione della strategia a livello di EndPoint e di UseCase. A livello di BusinessLogic non è stato eseguito alcun test poiché risulterebbero ripetitivi.

```
@Test
public void testEntity(){
    Entity e1 = new Entity(name: "E1");
    Entity e2 = new Entity(name: "E2");

    assertTrue(!e1.equals(e2));
    e1 = new Entity(e2);
    assertTrue(e1.equals(e2));
    e1 = new Entity(name: "E2");
    assertTrue(e1.equals(e2));
}

@Test
public void testCoupling(){
    Entity e1 = new Entity(name: "E1");
    Entity e2 = new Entity(name: "E2");
    Coupling c = new Coupling(e1, e2, Type.CC, coc: .76f);
    Coupling c2 = new Coupling(new Entity(name: "E3"), new Entity(name: "E4"), Type.CC, coc: .5f);
    assertEquals(expected: false, c.equals(c2));
    c2 = new Coupling(e1, e2, Type.CQ, coc: 0.76f);
    assertEquals(expected: false, c.equals(c2));
    c2 = new Coupling(e1, e2, Type.CC, coc: 0.2f);
    assertTrue(!c.equals(c2));
    assertEquals(new Coupling(new Entity(name: "E1"), new Entity(name: "E2"), Type.CC, coc: .76f));
    c2 = new Coupling(e1, new Entity(name: "E3"), Type.CC, coc: .76f);
    assertTrue(!c.equals(c2));
    assertEquals(Type.CC, c.getType());
    assertEquals(e1, c.getSrcEntity());
    assertEquals(e2, c.getDestEntity());
    Entity e3 = new Entity(name: "E9");
    c.setSrcEntity(e3);
    assertEquals(e3, c.getSrcEntity());
}
```

```

@Test
public void testCoOccurrenceMatrix(){
    ArrayList<Coupling> cl = new ArrayList<Coupling>();
    ArrayList<Coupling> cl2 = new ArrayList<Coupling>();
    ArrayList<Entity> el = new ArrayList<Entity>();
    for(int i=0;i<20;i++)
        el.add(new Entity("E"+i));

    for(int i=0;i<10;i++){
        cl.add(new Coupling(el.get(i), el.get(i+10), Type.CC, i));
        cl2.add(new Coupling(el.get(i), el.get(i+10), Type.CQ, i));
    }
    CoOccurrenceMatrix cm = new CoOccurrenceMatrix(Type.CC, cl);
    CoOccurrenceMatrix cm2 = new CoOccurrenceMatrix(Type.CC, cl);
    CoOccurrenceMatrix cm3 = new CoOccurrenceMatrix(Type.CQ, cl2);
    assertTrue(cm.equals(cm2));
    assertTrue(!cm.equals(cm3));
    assertTrue(!cm.equals(obj: null));
    cm.addCoValue(new Coupling(el.get(3),el.get(10),Type.CC, coc: 0.25f));
    assertTrue(!cm.equals(cm2));
    Float x = cm.getValue(el.get(0), el.get(10));

    assertEquals(expected: 0, x.intValue());
    assertNull(cm.getValue(el.get(0),new Entity(name: "E10")));

    Entity z = new Entity(name: "E30");
    cm.addCoValue(Type.CC, el.get(0), z, value: 0.5f);

    float x1 = (float) cm.getValue(el.get(0), z);
    assertEquals(expected: 0.5,x1 , delta: 0.00005f);

    Entity z1 = new Entity(name: "E50");
    Entity y1 = new Entity(name: "E60");
    cm.addCoValue(Type.CC, z1,y1,value: 0.7f);
    assertEquals(expected: 0.7f,(float) cm.getValue(z1, y1), delta: 0.00005);
    cm.addCoValue(Type.CC, y1,z1,value: 0.9f);
    assertEquals(expected: 0.9f,(float) cm.getValue(y1, z1), delta: 0.00005);
}

```

```

@Test
public void testEndPoint(){
    ArrayList<Coupling> cl = new ArrayList<Coupling>();
    ArrayList<Entity> el = new ArrayList<Entity>();
    for(int i=0;i<20;i++)
        el.add(new Entity("E"+i));

    for(int i=0;i<10;i++){
        cl.add(new Coupling(el.get(i), el.get(i+10), Type.CC, i));
        cl.add(new Coupling(el.get(i+10), el.get(i), Type.CC, i+10));
    }
    cl.add(new Coupling(el.get(0), el.get(10), Type.CC, freq: 100));
    ApplicationAbstraction aa = new EndPoint(cl, ID: "EP1", freq: 0.5f);
    assertNull(aa.getMapper().get(Type.CC));
    cl.remove(cl.size()-1);
    aa = new EndPoint(cl, ID: "EP1", freq: 0.5f);

    assertEquals(expected: 0.5, aa.getFrequency(), delta: 0.0005);
    assertEquals(expected: "EP1", aa.getID());

    assertEquals(expected: 5, (float) aa.getMapper().get(Type.CC).getValue(el.get(5), el.get(15)), delta: 0.00005);
    assertEquals(expected: 15, (float) aa.getMapper().get(Type.CC).getValue(el.get(15), el.get(5)), delta: 0.00005);
    assertEquals(expected: 0, (float) aa.getMapper().get(Type.CC).getValue(el.get(0), el.get(10)), delta: 0.00005);
    assertEquals(expected: 10, (float) aa.getMapper().get(Type.CC).getValue(el.get(10), el.get(0)), delta: 0.00005);

    ApplicationAbstraction aa2 = new EndPoint(cl, ID: "EP1", freq: 0.5f);
    assertTrue(aa.equals(aa2));
    assertTrue(!aa.equals(null));
}

```

```

@Test
public void testAverage(){
    ArrayList<ApplicationAbstraction> epList = new ArrayList<>();
    ArrayList<Coupling> cl = new ArrayList<Coupling>();
    ArrayList<Entity> el = new ArrayList<Entity>();
    for(int i=0;i<20;i++){
        el.add(new Entity("E"+i));
    for(int i=0;i<10;i++){
        cl.add(new Coupling(el.get(i), el.get(i+10), Type.CC, i+1));
        cl.add(new Coupling(el.get(i+10), el.get(i), Type.CC, (i+1)*100));
    }
    epList.add(new EndPoint(cl, ID: "EP1", freq: 0.5f));
    epList.add(new EndPoint(cl, ID: "EP2", freq: 0.25f));

    epList.get(0).buildMatrices();
    epList.get(1).buildMatrices();

    BuildCoMatStrategy bs = new Average();
    ArrayList<CoOccurrenceMatrix> ms = bs.buildCoMat(epList);
    float value,value1;
    for(int i = 0; i<10; i++){
        value = (float) ((i+1)*0.5 + (i+1)*0.25)/2f;
        value1 = (float) ((i+1)*0.5*100 + (i+1)*0.25*100)/2f;
        assertEquals(value, ms.get(0).getValue(el.get(i), el.get(i+10)), delta: 0.00001f);
        assertEquals(value1, ms.get(0).getValue(el.get(i+10),el.get(i)),delta: 0.00001f);
    }
}

```

```

    @Test
    public void testUseCase(){
        ArrayList<EndPoint> epList = new ArrayList<>();
        ArrayList<Coupling> cl = new ArrayList<Coupling>();
        ArrayList<Coupling> cl2 = new ArrayList<Coupling>();
        ArrayList<Entity> el = new ArrayList<Entity>();
        for(int i=0;i<20;i++)
            |   el.add(new Entity("E"+i));
        for(int i=0;i<10;i++){
            cl.add(new Coupling(el.get(i), el.get(i+10), Type.CC, i+1));
            cl.add(new Coupling(el.get(i+10), el.get(i), Type.CC, (i+1)*100));
            cl2.add(new Coupling(el.get(i), el.get(i+10), Type.CC, (i+1)*1000));
        }

        epList.add(new EndPoint(cl, ID: "EP1", freq: 0.5f));
        epList.add(new EndPoint(cl, ID: "EP2", freq: 0.25f));
        epList.add(new EndPoint(cl2, ID: "EP2", freq: 0.15f));
        epList.add(new EndPoint(cl2, ID: "EP2", freq: 0.15f));
        for(int i = 0; i< epList.size(); i++)
            |   epList.get(i).buildMatrices();
        UseCase uc = new UseCase(id: "UC1", f: 1, epList);
        uc.setStrategy(new Average());
        uc.buildMatrices();
        CoOccurrenceMatrix cocm = uc.getMapper().get(Type.CC);
        float value,value1;
        for(int i = 0; i<10; i++){
            value = (float) ((i+1)*0.5 + (i+1)*0.25 + (i+1)*1000f*0.15f)/3f;
            value1 = (float) ((i+1)*0.5*100 + (i+1)*0.25*100)/3f;
            assertEquals(value, cocm.getValue(el.get(i), el.get(i+10)), delta: 0.00001f);
            assertEquals(value1, cocm.getValue(el.get(i+10),el.get(i)),delta: 0.00001f);
        }
        UseCase uc2 = new UseCase(id: "UC1", f: 1, epList);
        assertTrue(!uc.equals(uc2));
        assertTrue(!uc.equals(obj: null));
        UseCase uc3 = new UseCase(id: "UC1", f: 1, epList, new Average());
        assertTrue(uc.equals(uc3));
    }
}

```

WeightedGraphTest

Classe di test per il grafo pesato. Essendo una struttura abbastanza complessa, è stato valutato utile eseguire dei test su queste classi.

```

@Test
public void testVertex(){
    Entity e1 = new Entity(name: "e1");
    Entity e2 = new Entity(name: "e2");

    Vertex v1 = new Vertex(e1);
    Vertex v2 = new Vertex(e2);

    assertEquals(expected: false, v1.equals(v2));
    v1 = new Vertex(e2);
    assertEquals(expected: true, v1.equals(v2));
}

@Test
public void testEdge(){

    ArrayList<Entity> enList = new ArrayList<Entity>();
    ArrayList<Vertex> vxList = new ArrayList<Vertex>();
    ArrayList<Edge> edList = new ArrayList<Edge>();

    for(int i=0;i<10;i++){
        enList.add(new Entity("en"+i));
        vxList.add(new Vertex(enList.get(i)));
    }
    for (int i=0; i<5; i++)
        edList.add(new Edge(i+1, vxList.get(i), vxList.get(i+5)));

    assertEquals(expected: true, edList.get(0).getConnVertex(vxList.get(0)).equals(vxList.get(5)));
    assertEquals(expected: true, edList.get(0).getConnVertex(vxList.get(5)).equals(vxList.get(0)));
    assertNull(edList.get(0).getConnVertex(vxList.get(1)));

    assertEquals(expected: false, edList.get(0).equals(edList.get(1)));
    Edge e = new Edge(weight: 1, vxList.get(5), vxList.get(0));
    assertEquals (expected: true, edList.get(0).equals(e));
    Edge e1 = new Edge (weight: 2, vxList.get(5), vxList.get(0));
    assertEquals (expected: false, e.equals(e1));
}

```

```

@Test
public void testGraph(){
    Graph g = new Graph();

    ArrayList<Vertex> vxList = new ArrayList<Vertex>();
    ArrayList<Edge> edList = new ArrayList<Edge>();

    for(int i=0;i<4;i++){
        vxList.add(new Vertex(new Entity("en"+i)));
        g.addVertex(vxList.get(i));
    }

    assertEquals(vxList.get(0), g.getVertex(new Entity (name: "en0")));

    for (int i=0; i<vxList.size(); i++){
        for(int j=0; j<vxList.size(); j++){
            if(i!=j)
                edList.add(new Edge(weight: 1, vxList.get(i), vxList.get(j)));
        }
    }

    for (Edge e : edList)
        g.addEdge(e);

    assertEquals(expected: 6, g.getEdgeList().size());
    assertEquals(expected: 4, g.getVertexList().size());

    int eSize = g.getEdgeList().size();
    int vSize = g.getVertexList().size();

    g.addVertex(new Vertex(new Entity(name: "en0")));
    g.addEdge(new Edge(weight: 1, vxList.get(1), vxList.get(0)));
    g.addEdge(new Edge(weight: 1, vxList.get(0), vxList.get(0)));
    g.addEdge(new Edge(weight: 1, vxList.get(0), new Vertex(new Entity(name: "entity"))));

    assertEquals(vSize, g.getVertexList().size());
    assertEquals(eSize, g.getEdgeList().size());

    g.removeEdge(edList.get(0));

    assertEquals(eSize-1, g.getEdgeList().size());
    g.removeVertex(vxList.get(0));

    assertFalse(g.getVertexList().contains(vxList.get(0)));
    assertEquals(eSize-3, g.getEdgeList().size());
}

```

ManagerTest

Classe di test per tutta la parte di creazione del grafo e semplificazione del grafo. Test molto importanti in quanto verificano la correttezza matematica delle varie strategie utilizzate.

```
@Test
public void testGraphManager() {
    GraphManager manager = new GraphManager();

    ArrayList<Entity> enList = new ArrayList<>();
    for (int i=0; i<4; i++){
        enList.add(new Entity("en"+i));
    }

    DomainModel dm = new DomainModel(enList);
    manager.setDomainModel(dm);

    ArrayList<Coupling> cpList = new ArrayList<>();
    for (Entity r : enList){
        for (Entity c : enList){
            if (!r.equals(c)){
                switch (r.getName()){
                    case "en0":
                        cpList.add(new Coupling(r, c, Type.CC, coc: 1.f));
                        cpList.add(new Coupling(r, c, Type.CQ, coc: 1.f));
                        cpList.add(new Coupling(r, c, Type.QC, coc: 1.f));
                        cpList.add(new Coupling(r, c, Type.QQ, coc: 1.f));
                        break;

                    case "en1":
                        cpList.add(new Coupling(r, c, Type.CC, coc: 0.5f));
                        cpList.add(new Coupling(r, c, Type.CQ, coc: 0.5f));
                        cpList.add(new Coupling(r, c, Type.QC, coc: 0.5f));
                        cpList.add(new Coupling(r, c, Type.QQ, coc: 0.5f));
                        break;

                    case "en2":
                        cpList.add(new Coupling(r, c, Type.CC, coc: 0.25f));
                        cpList.add(new Coupling(r, c, Type.CQ, coc: 0.25f));
                        cpList.add(new Coupling(r, c, Type.QC, coc: 0.25f));
                        cpList.add(new Coupling(r, c, Type.QQ, coc: 0.25f));
                        break;

                    default: break;
                }
            }
        }
    }

    ApplicationAbstraction ep = new EndPoint(cpList, ID: "ep", freq: 1.f);
    ep.buildMatrices();

    manager.setApplicationAbstraction(ep);
    manager.setWeightCC(wCC: 1.f);
    manager.setWeightCQ(wCQ: 0.75f);
    manager.setWeightQC(wQC: 0.5f);
    manager.setWeightQQ(wQQ: 0.25f);
}
```

```
Graph g = manager.getGraph();

assertEquals(expected: 4, g.getVertexList().size());
assertEquals(expected: 6, g.getEdgeList().size());

Edge e01 = g.getEdge(g.getVertexList().get(0), g.getVertexList().get(1));
Edge e23 = g.getEdge(g.getVertexList().get(2), g.getVertexList().get(3));
Edge e03 = g.getEdge(g.getVertexList().get(0), g.getVertexList().get(3));

assertEquals(expected: 0.75f, e01.getWeight(), delta: 0.00005);
assertEquals(expected: 0.125f, e23.getWeight(), delta: 0.00005);
assertEquals(expected: 0.5f, e03.getWeight(), delta: 0.00005);

}
```

```

@Test
public void testKruskal(){
    MSTClustering swk = new MSTClustering();
    ArrayList<Vertex> vl = new ArrayList<>();
    ArrayList<Edge> el = new ArrayList<>();
    for(int i = 0; i<5; i++){
        vl.add(new Vertex(new Entity("E"+i)));
    }
    Edge e1 = new Edge(weight: 0.15f, vl.get(0), vl.get(1));
    Edge e2 = new Edge(weight: 0.7f, vl.get(0), vl.get(2));
    Edge e3 = new Edge(weight: 0.3f, vl.get(2), vl.get(3));
    Edge e4 = new Edge(weight: 0.1f, vl.get(1), vl.get(3));
    Edge e5 = new Edge(weight: 0.5f, vl.get(1), vl.get(4));
    Edge e6 = new Edge(weight: 0.2f, vl.get(3), vl.get(4));
    el.add(0,e1);
    el.add(1,e2);
    el.add(2,e3);
    el.add(3,e4);
    el.add(4,e5);
    el.add(5,e6);
    Graph g = new Graph();
    for(Vertex v : vl)
        g.addVertex(v);
    for(Edge e : el)
        g.addEdge(e);
    Graph sg = swk.Kruskal(g);

    assertEquals(expected: false, sg.getEdgeList().contains(e1));
    assertEquals(expected: true, sg.getEdgeList().contains(e2));
    assertEquals(expected: true, sg.getEdgeList().contains(e3));
    assertEquals(expected: false, sg.getEdgeList().contains(e4));
    assertEquals(expected: true, sg.getEdgeList().contains(e5));
    assertEquals(expected: true, sg.getEdgeList().contains(e6));

    assertEquals(e2, sg.getVertexList().get(0).getNeighbour().get(0));
    assertEquals(expected: 1,sg.getVertexList().get(0).getNeighbour().size());

    assertEquals(e2, sg.getVertexList().get(2).getNeighbour().get(0));
    assertEquals(expected: 2,sg.getVertexList().get(2).getNeighbour().size());

    assertEquals(e3, sg.getVertexList().get(2).getNeighbour().get(1));
    assertEquals(expected: 2,sg.getVertexList().get(2).getNeighbour().size());

    assertEquals(e3, sg.getVertexList().get(3).getNeighbour().get(0));
    assertEquals(expected: 2,sg.getVertexList().get(3).getNeighbour().size());

    assertEquals(e5, sg.getVertexList().get(1).getNeighbour().get(0));
    assertEquals(expected: 1,sg.getVertexList().get(1).getNeighbour().size());

    assertEquals(e5, sg.getVertexList().get(4).getNeighbour().get(0));
    assertEquals(expected: 2,sg.getVertexList().get(4).getNeighbour().size());

    assertEquals(e6, sg.getVertexList().get(3).getNeighbour().get(1));
    assertEquals(expected: 2,sg.getVertexList().get(3).getNeighbour().size());

    assertEquals(e6, sg.getVertexList().get(4).getNeighbour().get(1));
    assertEquals(expected: 2,sg.getVertexList().get(4).getNeighbour().size());
}

```

```

@Test
public void simplifyWithKruskal(){
    MSTClustering swk = new MSTClustering(n: 4, s: 3);
    ArrayList<Vertex> vl = new ArrayList<>();
    ArrayList<Edge> el = new ArrayList<>();
    for(int i = 0; i<10; i++){
        vl.add(new Vertex(new Entity("E"+i)));
    }

    el.add(new Edge(weight: 0.15f, vl.get(0), vl.get(1)));
    el.add(new Edge(weight: 0.5f, vl.get(0), vl.get(3)));
    el.add(new Edge(weight: 0.7f, vl.get(3), vl.get(1)));
    el.add(new Edge(weight: 0.3f, vl.get(1), vl.get(4)));
    el.add(new Edge(weight: 0.05f, vl.get(1), vl.get(2)));
    el.add(new Edge(weight: 0.8f, vl.get(2), vl.get(4)));
    el.add(new Edge(weight: 0.9f, vl.get(2), vl.get(5)));
    el.add(new Edge(weight: 0.1f, vl.get(3), vl.get(4)));
    el.add(new Edge(weight: 0.3f, vl.get(4), vl.get(5)));
    el.add(new Edge(weight: 0.01f, vl.get(4), vl.get(8)));
    el.add(new Edge(weight: 0.15f, vl.get(5), vl.get(8)));
    el.add(new Edge(weight: 0.1f, vl.get(5), vl.get(9)));
    el.add(new Edge(weight: 0.5f, vl.get(8), vl.get(9)));
    el.add(new Edge(weight: 0.3f, vl.get(7), vl.get(8)));
    el.add(new Edge(weight: 0.5f, vl.get(6), vl.get(7)));

    for(Vertex v : vl){
        for(Edge e : el){
            if(e.getVertex1().equals(v) || e.getVertex2().equals(v)){
                v.addEdge(e);
            }
        }
    }

    Graph g = new Graph();
    for(Vertex v : vl){
        g.addVertex(v);
    }
    for(Edge e : el){
        g.addEdge(e);
    }
    Graph sg = swk.simplifyGraph(g);

    assertEquals(expected: true, contains(sg, el.get(0)));
    assertEquals(expected: true, contains(sg, el.get(1)));
    assertEquals(expected: true, contains(sg, el.get(2)));
    assertEquals(expected: false, contains(sg, el.get(3)));
    assertEquals(expected: false, contains(sg, el.get(4)));
    assertEquals(expected: true, contains(sg, el.get(5)));
    assertEquals(expected: true, contains(sg, el.get(6)));
    assertEquals(expected: false, contains(sg, el.get(7)));
    assertEquals(expected: true, contains(sg, el.get(8)));
    assertEquals(expected: false, contains(sg, el.get(9)));
    assertEquals(expected: false, contains(sg, el.get(10)));
    assertEquals(expected: false, contains(sg, el.get(11)));
    assertEquals(expected: true, contains(sg, el.get(12)));
    assertEquals(expected: false, contains(sg, el.get(13)));
    assertEquals(expected: true, contains(sg, el.get(14)));
}

```

```

@Test
public void testLossFunction(){
    MSTClustering swk = new MSTClustering(n: 4, s: 3);
    ArrayList<Vertex> vl = new ArrayList<>();
    ArrayList<Edge> el = new ArrayList<>();
    for(int i = 0; i<10; i++){
        vl.add(new Vertex(new Entity("E"+i)));
    }

    el.add(new Edge(weight: 0.14f, vl.get(0), vl.get(1)));
    el.add(new Edge(weight: 0.5f, vl.get(0), vl.get(3)));
    el.add(new Edge(weight: 0.7f, vl.get(3), vl.get(1)));
    el.add(new Edge(weight: 0.25f, vl.get(1), vl.get(4)));
    el.add(new Edge(weight: 0.05f, vl.get(1), vl.get(2)));
    el.add(new Edge(weight: 0.8f, vl.get(2), vl.get(4)));
    el.add(new Edge(weight: 0.9f, vl.get(2), vl.get(5)));
    el.add(new Edge(weight: 0.1f, vl.get(3), vl.get(4)));
    el.add(new Edge(weight: 0.35f, vl.get(4), vl.get(5)));
    el.add(new Edge(weight: 0.01f, vl.get(4), vl.get(8)));
    el.add(new Edge(weight: 0.15f, vl.get(5), vl.get(8)));
    el.add(new Edge(weight: 0.12f, vl.get(5), vl.get(9)));
    el.add(new Edge(weight: 0.55f, vl.get(8), vl.get(9)));
    el.add(new Edge(weight: 0.3f, vl.get(7), vl.get(8)));
    el.add(new Edge(weight: 0.52f, vl.get(6), vl.get(7)));

    for(Vertex v : vl){
        for(Edge e : el){
            if(e.getVertex1().equals(v) || e.getVertex2().equals(v)){
                v.addEdge(e);
            }
        }
    }

    Graph g = new Graph();
    for(Vertex v : vl){
        g.addVertex(v);
    }
    for(Edge e : el){
        g.addEdge(e);
    }
    Graph sg = swk.simplifyGraph(g);
    LossFunctionStrategy sfs = new LossStrategy1();
    assertEquals(expected: 0.065333, sfs.lossFunction(g, sg), delta: 0.000001);
}

```

```

@Test
public void testMyBestSolutionKruskal() {
    MSTClustering swk = new MSTClustering(n: 3, s: 3);
    ArrayList<Vertex> vl = new ArrayList<>();
    ArrayList<Edge> el = new ArrayList<>();
    for(int i = 0; i<10; i++){
        vl.add(new Vertex(new Entity("E"+i)));
    }

    el.add(new Edge(weight: 0.15f, vl.get(0), vl.get(1)));
    el.add(new Edge(weight: 0.5f, vl.get(0), vl.get(3)));
    el.add(new Edge(weight: 0.7f, vl.get(3), vl.get(1)));
    el.add(new Edge(weight: 0.3f, vl.get(1), vl.get(4)));
    el.add(new Edge(weight: 0.05f, vl.get(1), vl.get(2)));
    el.add(new Edge(weight: 0.8f, vl.get(2), vl.get(4)));
    el.add(new Edge(weight: 0.9f, vl.get(2), vl.get(5)));
    el.add(new Edge(weight: 0.1f, vl.get(3), vl.get(4)));
    el.add(new Edge(weight: 0.3f, vl.get(4), vl.get(5)));
    el.add(new Edge(weight: 0.01f, vl.get(4), vl.get(8)));
    el.add(new Edge(weight: 0.15f, vl.get(5), vl.get(8)));
    el.add(new Edge(weight: 0.1f, vl.get(5), vl.get(9)));
    el.add(new Edge(weight: 0.5f, vl.get(8), vl.get(9)));
    el.add(new Edge(weight: 0.3f, vl.get(7), vl.get(8)));
    el.add(new Edge(weight: 0.5f, vl.get(6), vl.get(7)));

    for(Vertex v : vl){
        for(Edge e : el){
            if(e.getVertex1().equals(v) || e.getVertex2().equals(v)){
                v.addEdge(e);
            }
        }
    }

    Graph g = new Graph();
    for(Vertex v : vl){
        g.addVertex(v);
    }
    for(Edge e : el){
        g.addEdge(e);
    }

    LossFunctionStrategy sfs = new LossStrategy1();
    Graph sg = swk.myBestSolution(g, sfs);
    assertEquals(expected: 0.027333386, sfs.lossFunction(g, sg), delta: 0.0001);
}

private boolean contains(Graph g, Edge e){
    for(Edge e1 : g.getEdgeList()){
        if(e1.equals(e)){
            return true;
        }
    }
    return false;
}

```

Appendice

Come usare il programma

All'interno della cartella principale “*Refactoring*” è presente la cartella “*Input*”, in cui si può trovare delle sottocartelle, ognuna delle quali rappresenta un esempio di Input. Una volta avviato il programma, verrà richiesto all'utente di sceglierne uno tra quelli presenti nella cartella.

Ogni esempio di Input è scritto in json e dovrà contenere dei precisi file con dei precisi nomi:

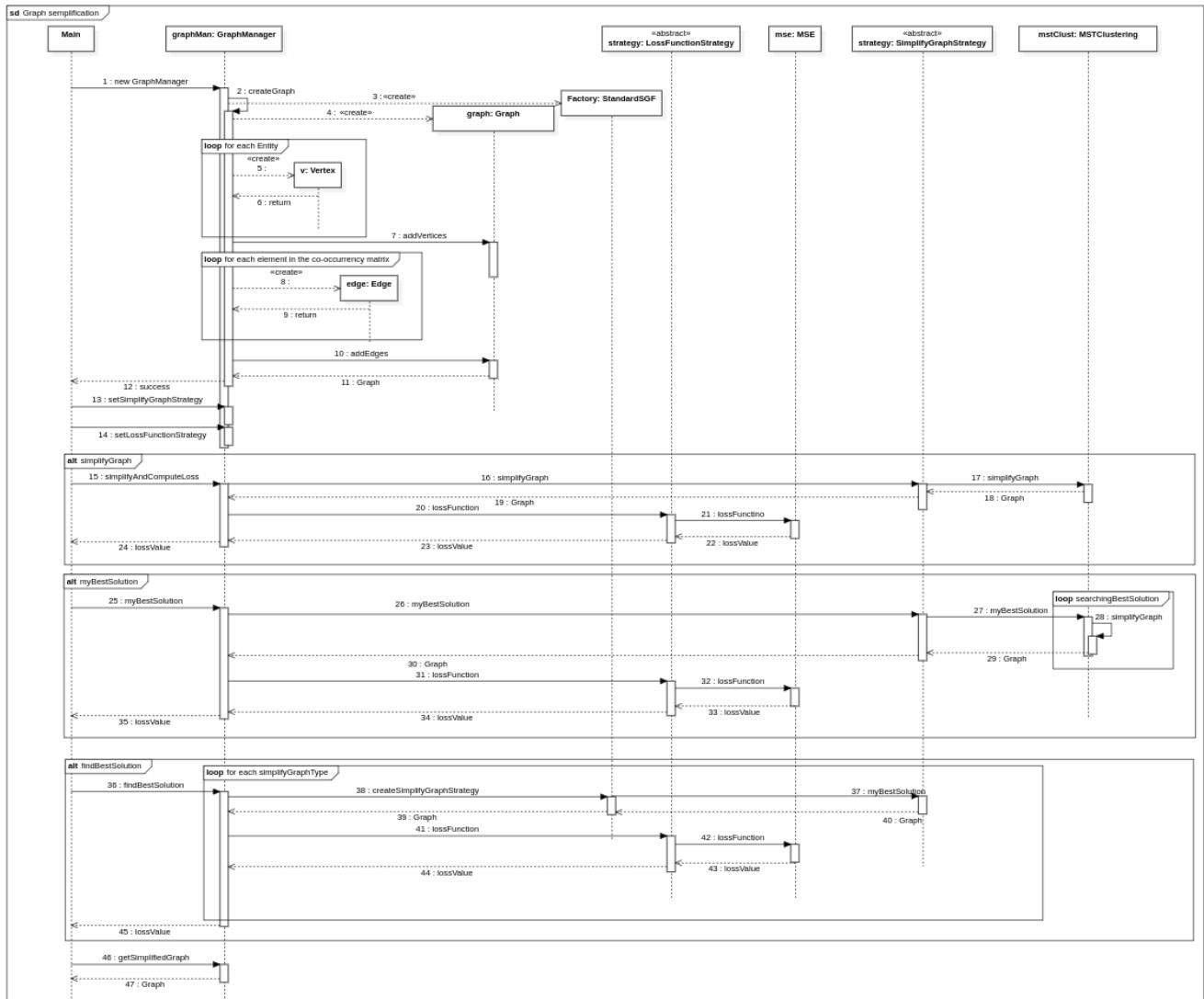
1. ***Entities.json*** contiene la lista dei nomi delle entità che costituiscono l'applicazione monolitica.
2. ***EndPoints.json*** contiene la lista degli EndPoint, ognuno dei quali contiene:
 - a. *ID* dell'EndPoint
 - b. *frequency*, cioè la frequenza di utilizzo dell'EndPoint
 - c. *coupList*, cioè la lista dei Coupling che compongono l'EndPoint, ognuno dei quali, a sua volta, è composto da:
 - i. *type*, cioè il tipo di co-occorrenza (CC, CQ, QC, QQ)
 - ii. *coOccurrence*, che descrive il valore effettivo della co-occorrenza
 - iii. *entity1* e *entity2*, cioè le due entità coinvolte nell'accoppiamento
3. ***UseCases.json*** contiene la lista degli Use Case, definiti nel seguente modo:
 - a. *ID* dello Use Case
 - b. *frequency*, cioè la frequenza di utilizzo dello Use Case
 - c. *buildMatStrategy*, in questo campo si specifica la strategia (appartenente a *BuildCoMatStrategy*) che si vuole utilizzare per la costruzione delle Matrici di Cooccorrenza di livello Use Case
 - d. *endPointList*, lista dei nomi degli EndPoint che costituiscono lo Use Case
4. ***BusinessLogic.json*** contiene le informazioni riguardanti la BusinessLogic, tra cui:
 - a. *ID* della BusinessLogic
 - b. *buildMatStrategy*, in questo campo si specifica la strategia (appartenente a *BuildCoMatStrategy*) che si vuole utilizzare per la costruzione delle Matrici di Cooccorrenza di livello BusinessLogic
5. ***GraphManagerSettings.json*** contiene impostazioni aggiuntive importanti che permettono all'utente di scegliere quali algoritmi utilizzare per arrivare al risultato finale, cioè:

- a. *simplifyGraphStrategy*, qui si setta la strategia di semplificazione del grafo desiderata, scegliendo la ConcreteStrategy tra quelle presenti nel SimplifyGraphStrategy.
- b. *lossFunctionStrategy*, specifica la funzione di Loss che si vuole utilizzare per confrontare diverse soluzioni di semplificazione del grafo. Si sceglie tra quelle presenti in LossFunctionStrategy.
- c. *weightCC*, *weightCQ*, *weightQC*, *weightQQ*, sono i pesi che si vuole attribuire alla diverse matrici di co-occorrenza. Verranno utilizzati per costruire il grafo pesato.

Inoltre, a runtime, se si è scelto MSTClustering come strategia di semplificazione del grafo, verrà chiesto all'utente di specificare i due parametri *n* ed *s*, rispettivamente il numero di partizioni desiderate e il massimo numero di entità per partizione. Per partizione si intende sostanzialmente il microservizio.

Sequence Diagram

Di seguito riportiamo lo scenario di interazione applicazione/utente della parte di semplificazione del grafo.



Bibliografia

Mazlami, Genc, et al. "Extraction of microservices from monolithic software architectures."

IEEE Xplore, ICW, 2017, <https://ieeexplore.ieee.org/abstract/document/8029803>.

Accessed 10 10 2021.