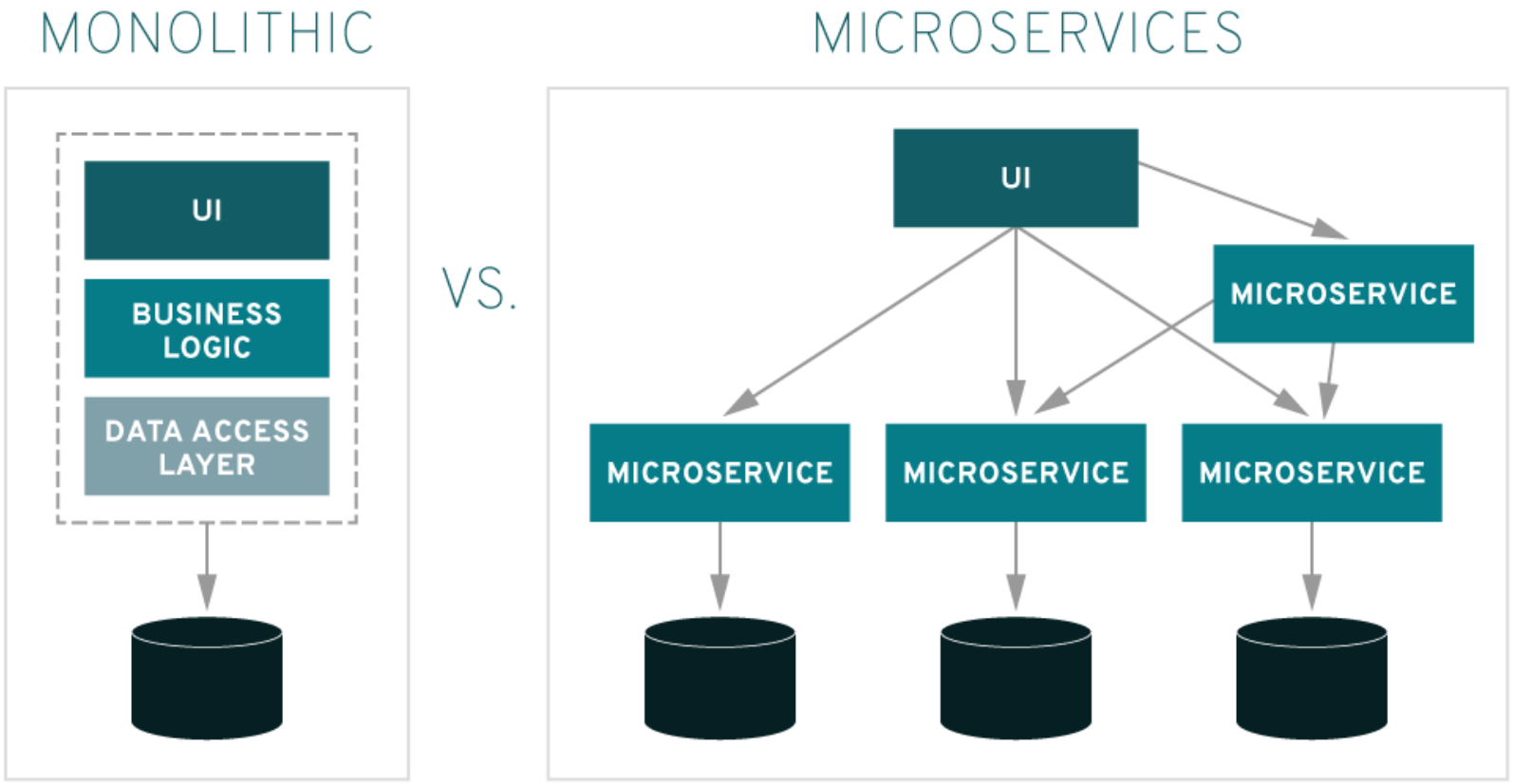
UNIVERSITÀ DEGLI STUDI FIRENZE
DINFO
Dipartimento di Ingegneria dell'Informazione

Refactoring di applicazioni Monolitiche

Federico Magini, Emanuele Nencioni, Giacomo Sodero

Obiettivo

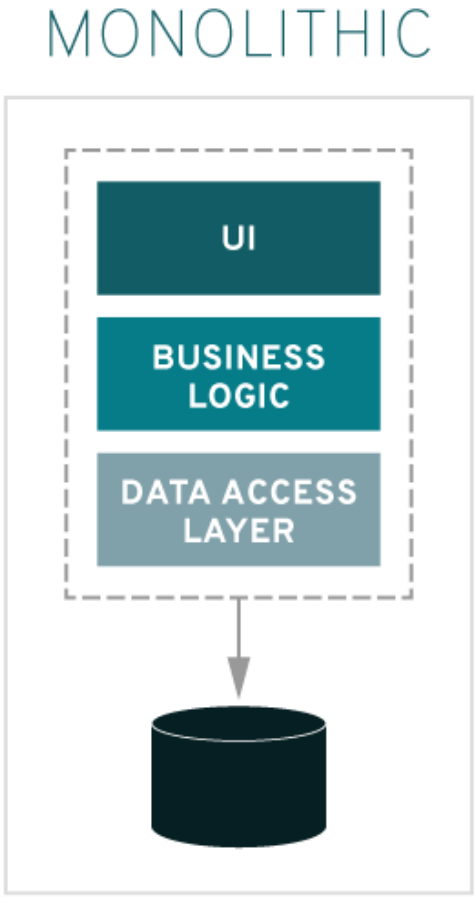
- L'obiettivo principale del progetto è quello di fare Refactoring di applicazioni monolitiche restful in forma di microservizi.



Applicazione Monolitica Restful

- Architettura che prevede data layer, business logic e domain model tutto rappresentato in un unico blocco.

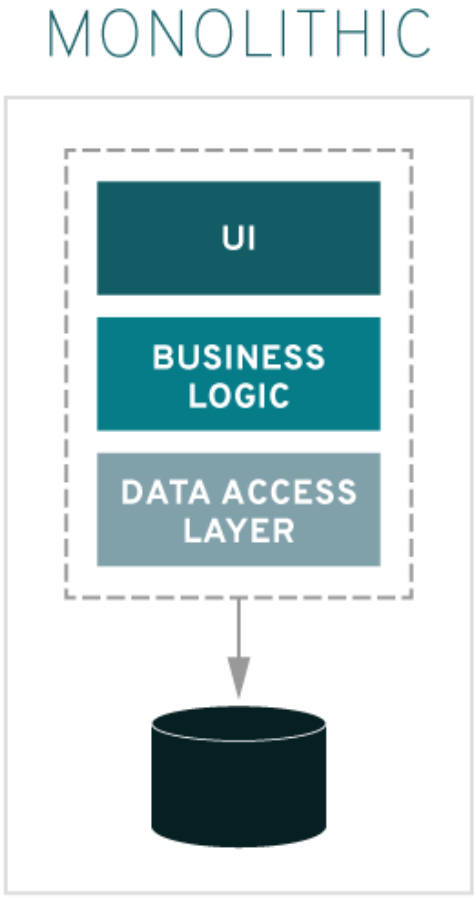
- Pregi:
  1. Più **rapido** ed efficiente nelle fasi di **test** e operazioni di **debug**
  2. **Architettura semplice** ed immediata
  3. **Bassa latenza** di comunicazione tra i componenti interni dell'applicazione
  4. **Sicurezza** agevolata dato che risiede tutto in un unico blocco
  5. Facile **distribuzione**, in quanto è possibile rilasciare in una sola volta tutti gli elementi che la costituiscono.



Applicazione Monolitica Restful

- Architettura che prevede data layer, business logic e domain model tutto rappresentato in un unico blocco.

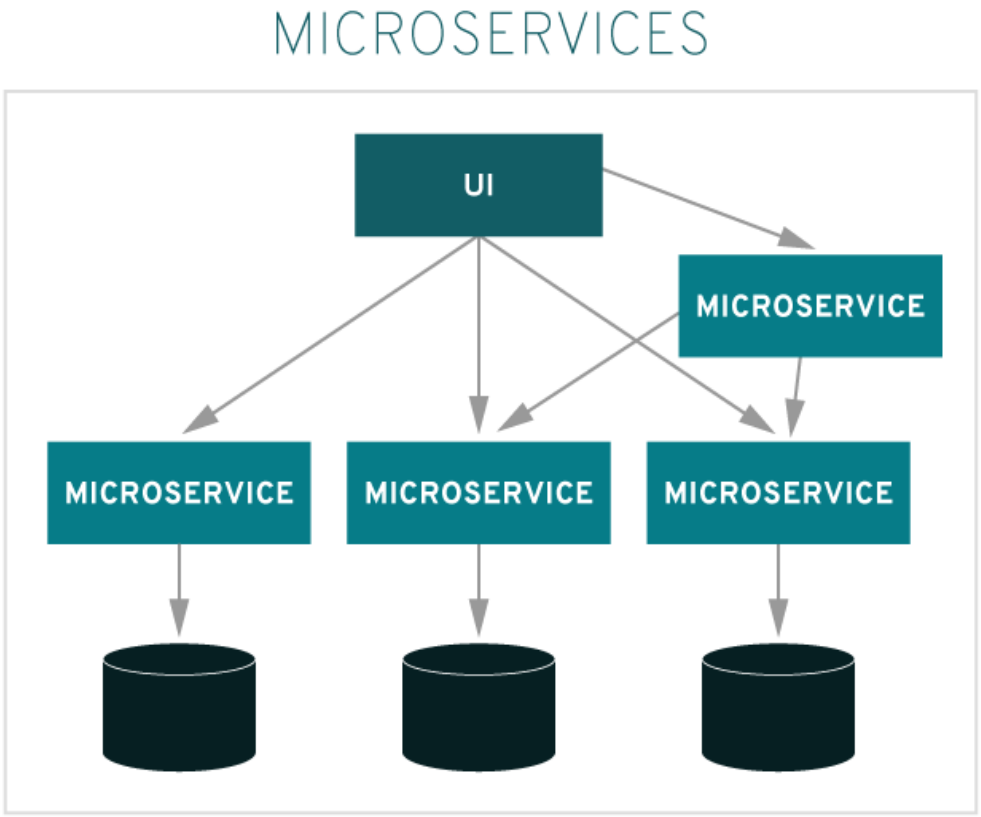
- Difetti:
  1. **Problemi** con applicazioni di **grandi dimensioni** e con alto numero di richieste.
  2. **Mancanza di modularità** che permetterebbe una facile espansione a nuove funzionalità o aggiornamenti (manutenzione scomoda)
  3. **Scarsa affidabilità**, in quanto la presenza di numerosi processi dipendenti e strettamente connessi tra loro, fa sì che la presenza di errori in una componente potrebbe influire sul funzionamento dell'intero software.
  4. Una minima modifica comporterebbe la **ridistribuzione** dell'intero monolite.



Applicazione a microservizi

- Architettura che si basa sul disaccoppiamento delle componenti in più blocchi indipendenti.

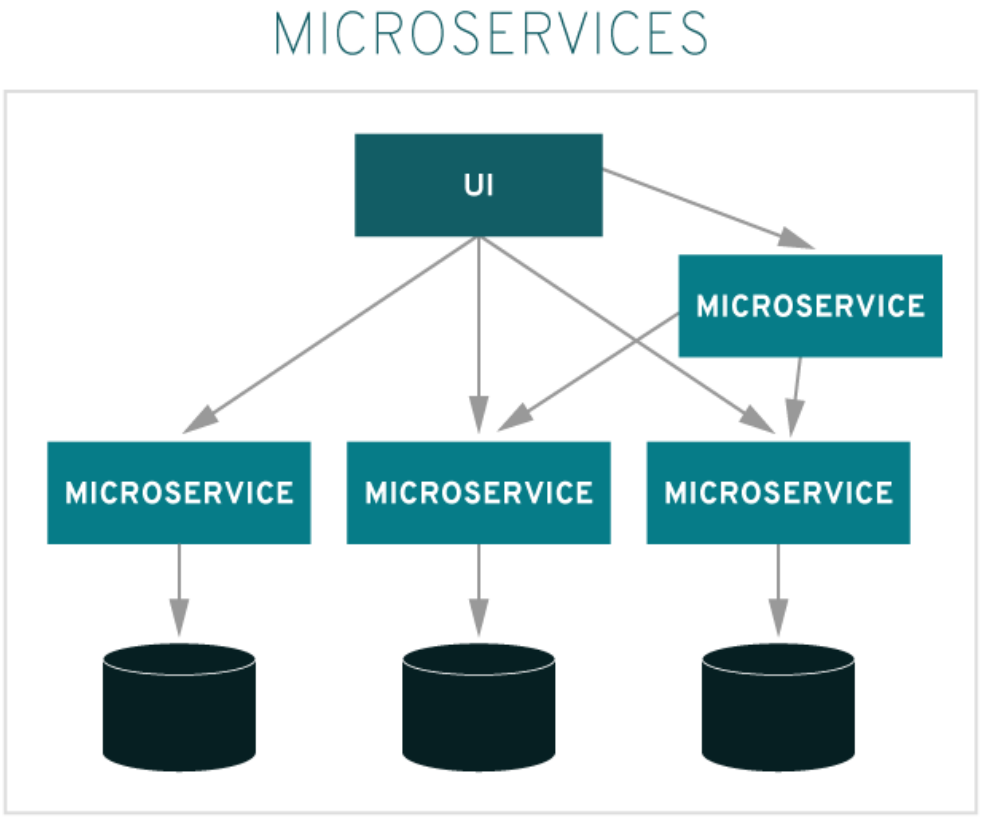
- Pregi
  1. **Unico scopo**: Ogni microservizio fornisce un unico servizio.
  2. **Loose coupling**: Accoppiamento ridotto per le entità in diversi microservizi.
  3. **Alta coesione**: Ogni microservizio racchiude tutti i metodi e le entità correlate. Questo implica facilità di manutenzione e espansione.



Applicazione a microservizi

- Architettura che si basa sul disaccoppiamento delle componenti in più blocchi indipendenti.

- Difetti
  1. **Sicurezza**: Mantenere un alto livello di sicurezza tra tutti i microservizi è un problema complesso.
  2. Problema quando si hanno delle stesse **entità** in diversi microservizi ed occorre mantenerle **allineate**.
  3. **Debug difficoltoso**: Ogni microservizio ha il suo set di input, ed inoltre potrebbe essere eseguito in modalità distribuita su più macchine.



COME FARE ?

- L'applicazione è divisa in due step:
  - Rappresentazione del Domain Model dell'applicazione monolitica come un grafo pesato
  - Semplificazione del grafo tramite tagli in modo da generare i diversi microservizi.

COME FARE ?

- Per la finalità del progetto, si ipotizza di avere in ingresso i seguenti dati:
  - **Entità** del Domain model.
  - **Accoppiamento** tra le varie entità.
  - Gli **endpoint** dell'applicazione monolitica descrivendo le entità che ne fanno parte.
  - **Use case** dell'applicazione monolitica descrivendo gli endpoint che ne fanno parte.

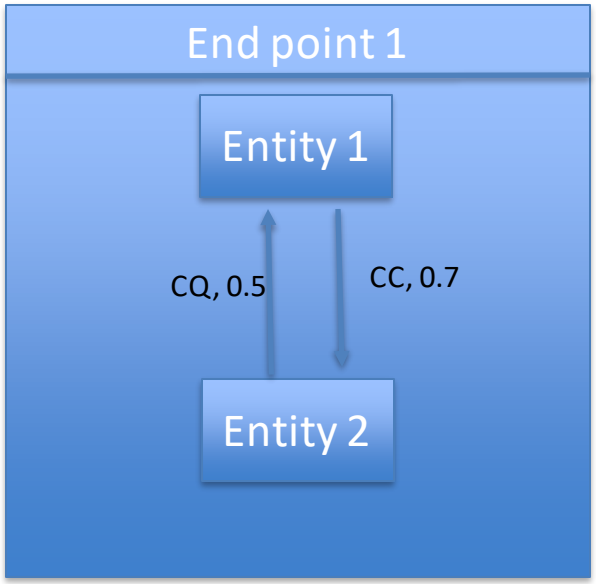
Le entità, i couplings, gli endpoints e gli UseCases, diventeranno classi della nostra applicazione.

COME FARE ?

- I dati degli accoppiamenti in ingresso verranno poi inseriti in oggetti di tipo EndPoint:
  - Si creano così delle matrici di co-occorrenza: Una matrice che racchiude tutti i coupling delle entità che vengono usate da un particolare end point.

- Siccome tra due entità si hanno 4 tipi di coupling diverso(Command-Command, Command-Query, Query-Command, Query-Query), avremo quindi 4 tipi di matrici di co-occorrenza.

Esempio di matrici di co-occorrenza



$p(Command\ on\ E1\ |Command\ on\ E2)$

CC	Entity 1	Entity 2
Entity 1	\	0
Entity 2	0.7	\

CQ	Entity 1	Entity 2
Entity 1	\	0.5
Entity 2	0	\

$p(Command\ on\ E1\ |Query\ on\ E2)$

- Le matrici non sono simmetriche poiché in generale (vale per tutti e 4 i tipi di coupling):

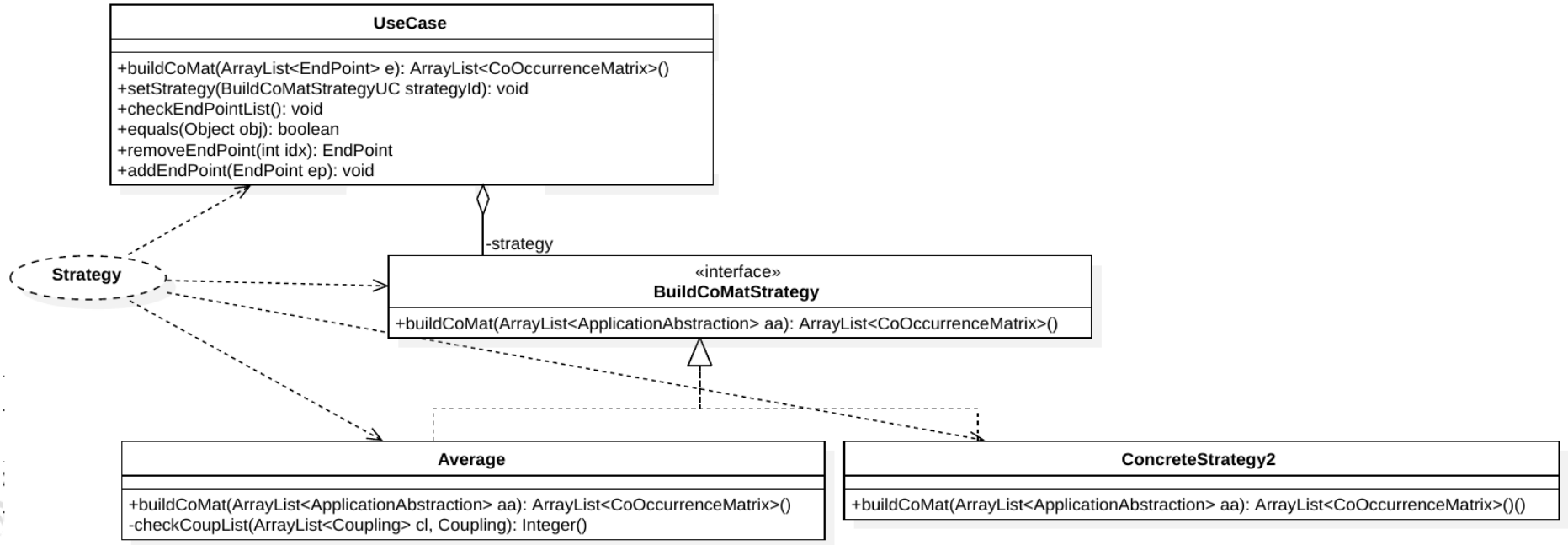
$$p(Command\ on\ E2\ |Command\ on\ E1) \neq\ p(Command\ on\ E1\ |Command\ on\ E2)$$

Da EndPoint a UseCase

- Questo concetto di matrici di co-occorrenza si può applicare anche a livello di Use Case dell'applicazione monolitica:
  - Una volta istanziati tutti gli oggetti di tipo EndPoint con le relative matrici di co-occorrenza, si possono creare tutti i vari oggetti di tipo UseCase ripetendo il procedimento.
    Questa volta però si dovranno unire diverse matrici di co-occorrenza nelle quattro che rappresenterebbero lo UseCase.

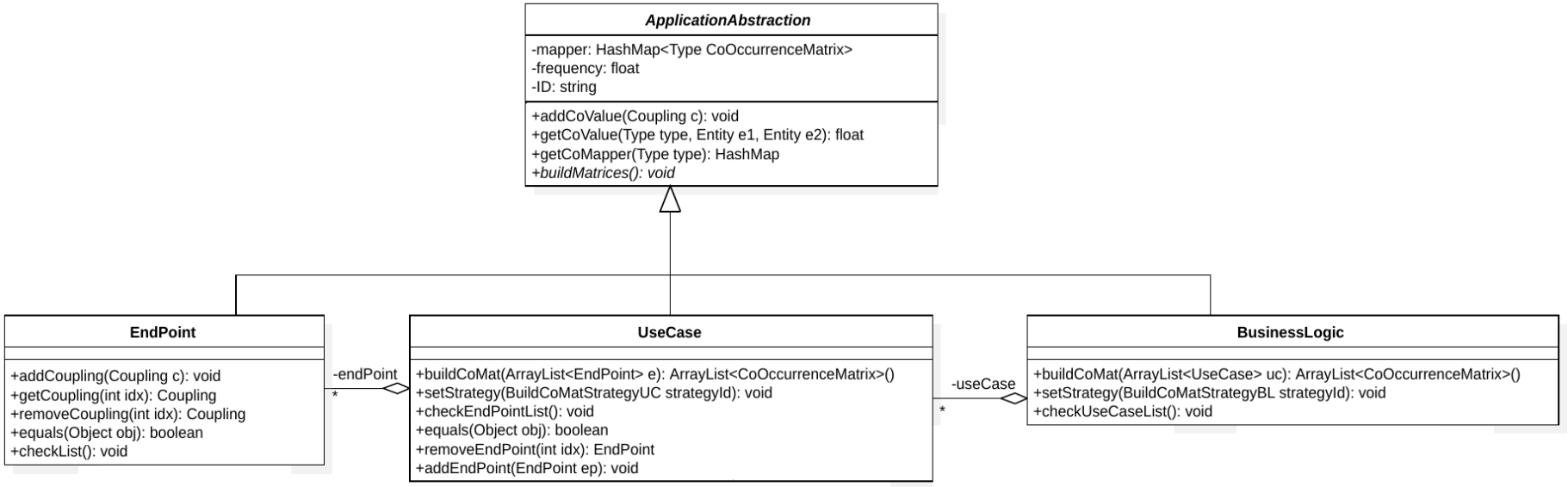
Da EndPoint a UseCase

- Potrebbero esserci vari modi per costruire le matrici di co-occorrenza da altre matrici preesistenti, perciò la soluzione migliore è aggiungere un aspetto modulare a questa funzionalità inserendo uno **Strategy pattern**:



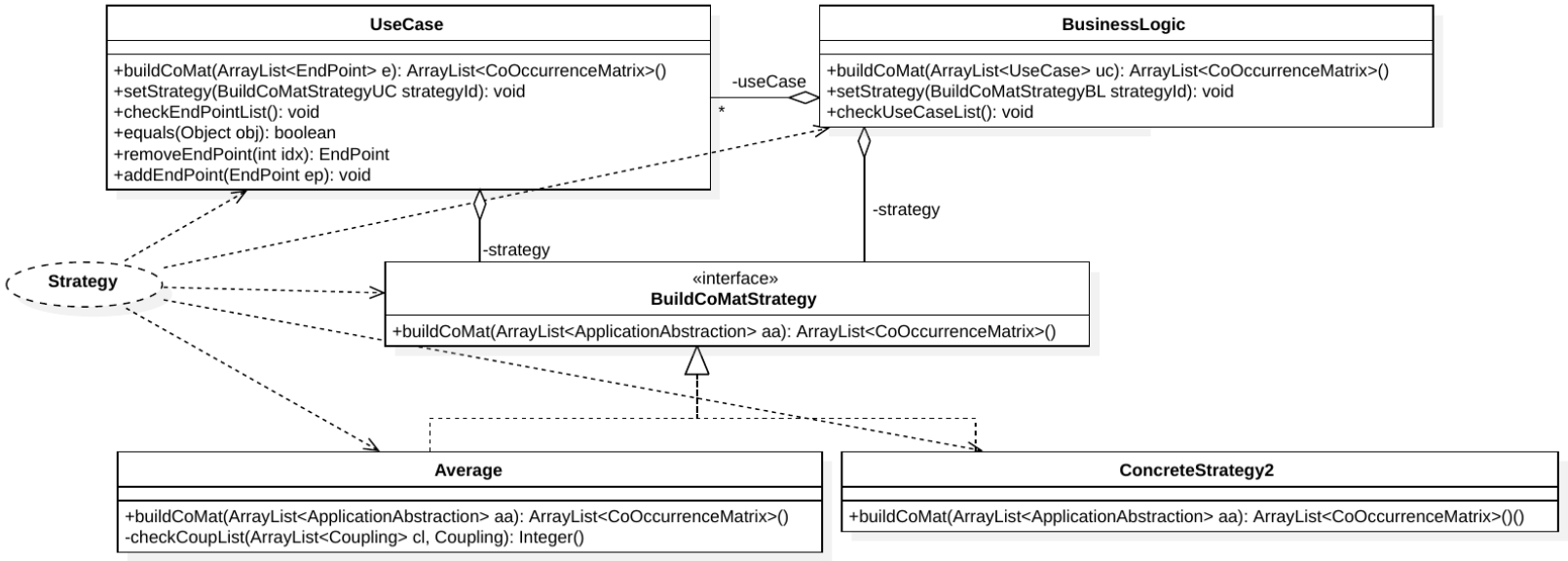
ApplicationAbstraction

- Ogni UseCase fa parte di una stessa BusinessLogic, quindi si può eseguire gli stessi step fatti per UseCase salendo di livello.
  - Si è fornita a UseCase, EndPoint e BusinessLogic questa struttura:

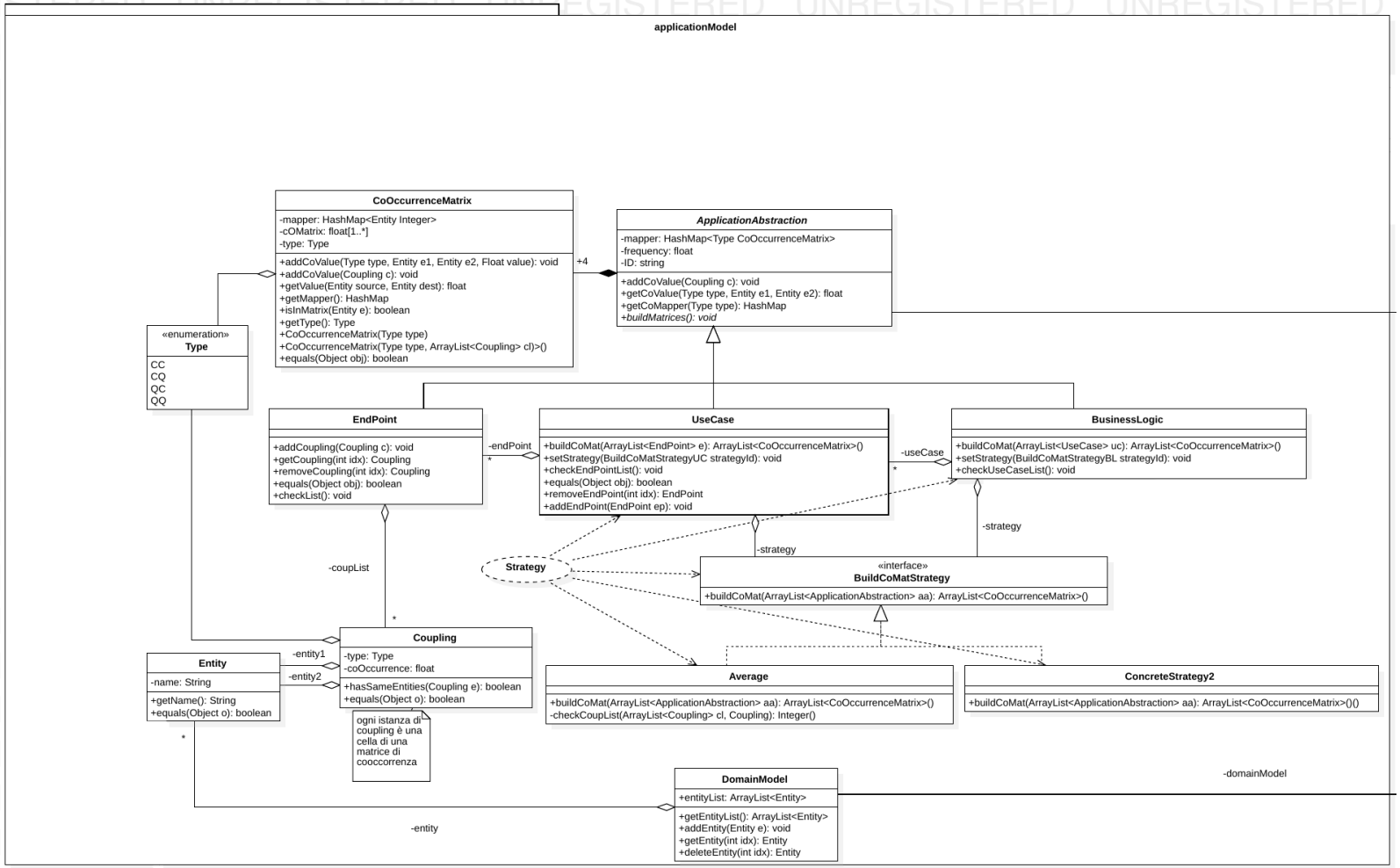


Strategy completo

- In questo modo si può applicare anche alla classe BusinessLogic lo strategy pattern prima descritto (dato che adesso entrambi hanno la stessa interfaccia per le matrici di co-occorrenza):



ApplicationModel

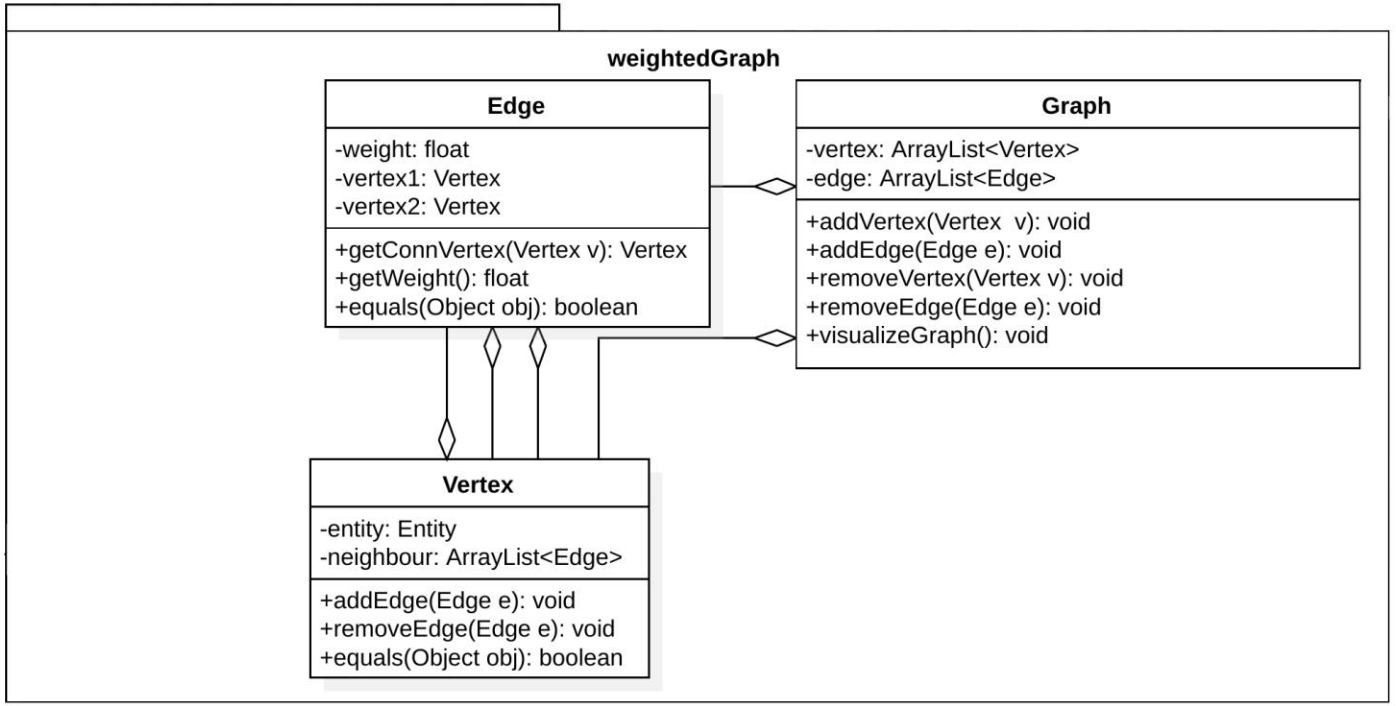


Grafo

- Adesso si hanno a disposizione 4 matrici di co-occorrenza che descrivono l'accoppiamento delle varie entità dell'applicazione monolitica.
- Lo step successivo è quello di modellare il domain model come un grafo pesato:
  - Vertici: Entità dell'applicazione
  - Archi: accoppiamento tra due entità. Il peso corrisponde al valore dell'accoppiamento.

Grafo

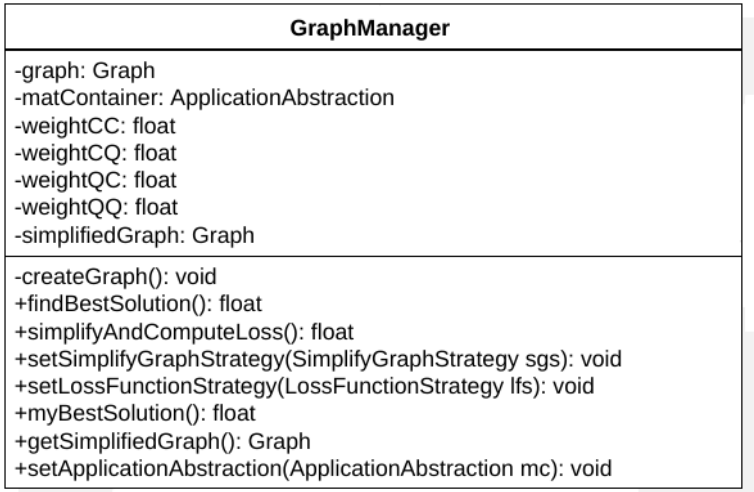
Di seguito si può vedere la struttura del grafo pesato ideata.



Grafo

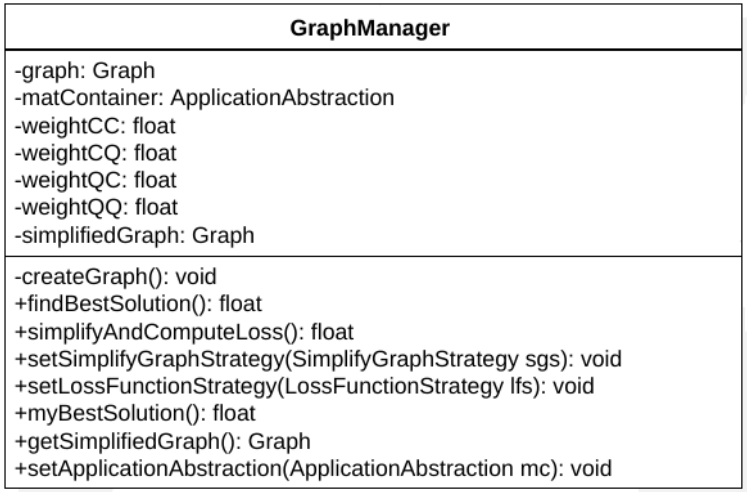
- Prima di tutto occorre unire le 4 matrici di co-occorrenza in una sola. Questa sarà poi trasformata in un grafo e successivamente si potrà eseguire un processo di semplificazione del grafo che ci porterà ad avere un grafo con più componenti connesse.
- Ognuna delle componenti connesse rappresenterà un microservizio.
- Per creare il grafo e gestirlo, è stata fatta una classe ad hoc: GraphManager

Grafo



- Questa classe permette di creare il grafo partendo da qualsiasi ApplicationAbstraction con matrici di co-occorrenza istanziate. Questo permetterebbe pure di semplificare non solo un oggetto di tipo BusinessLogic, ma anche oggetti di tipo EndPoint o UseCase.

Grafo



- Per creare il grafo dell'applicazione monolitica, si calcola la media ponderata dei valori nelle matrici di co-occorrenza per ogni coppia di entità:

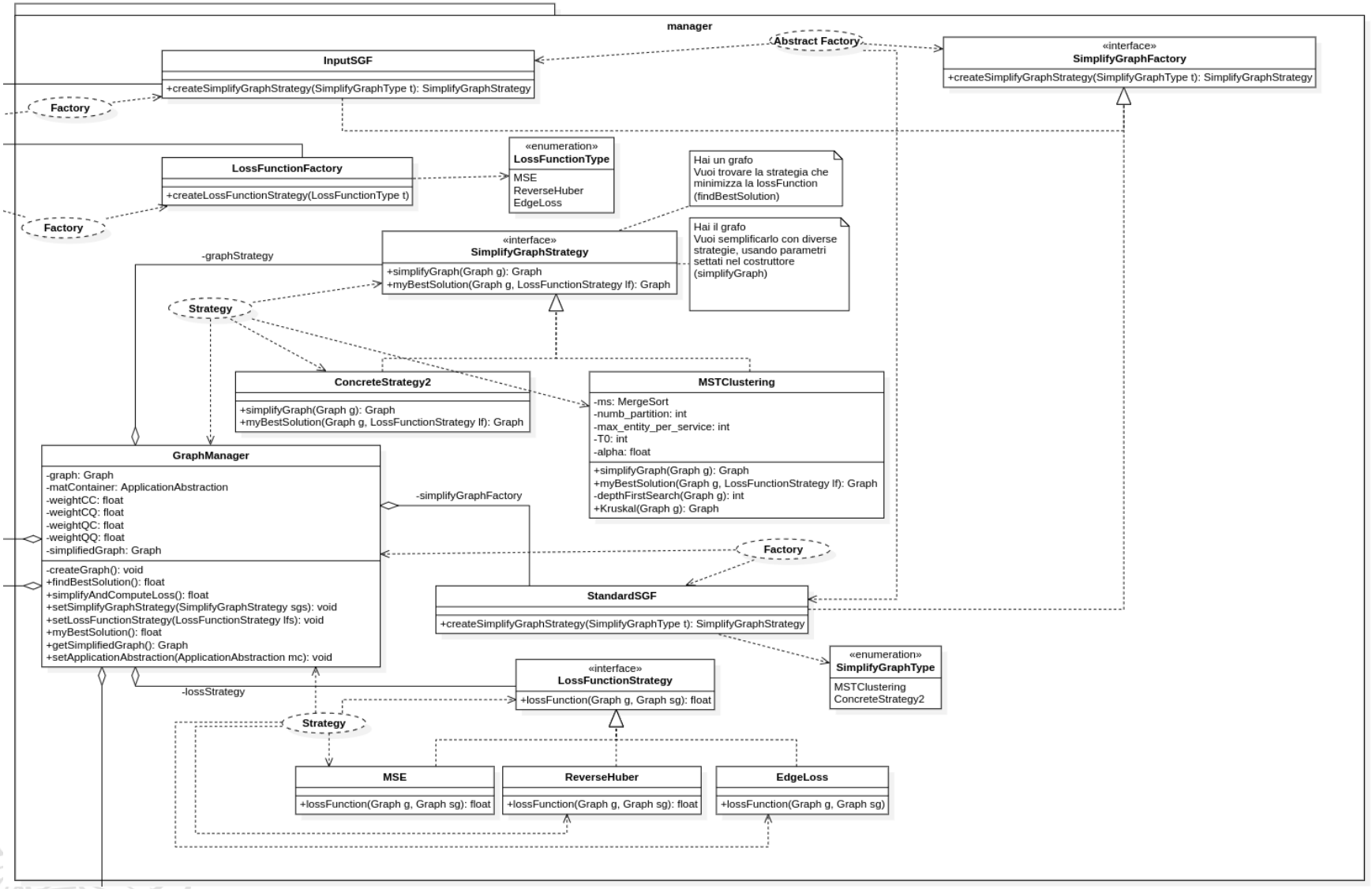
$$E_{rc} = \frac{\sum_{i,j \,\epsilon\, I} w_{ij}\left(M_{ij}[r][c] + M_{ij}[c][r]\right)}{2w_{cc} + 2w_{cq} + 2w_{qc} + 2w_{qq}}$$

- $I = \{c, q\}$
- M_{ij} rappresenta una matrice di co-occorrenza
- w_{ij} peso di un tipo di co-occorrenza
- E_{rc} arco indiretto tra vertice r e vertice c, $E_{rc} = E_{cr}$.

Semplificazione del grafo

- A questo punto si ha un grafo indiretto che rappresenta l'intera applicazione monolitica.
- Per cercare di estrarre da questo grafo dei microservizi, si cerca di disaccoppiare dal grafo le entità che sono meno connesse tra loro.

Package: Manager

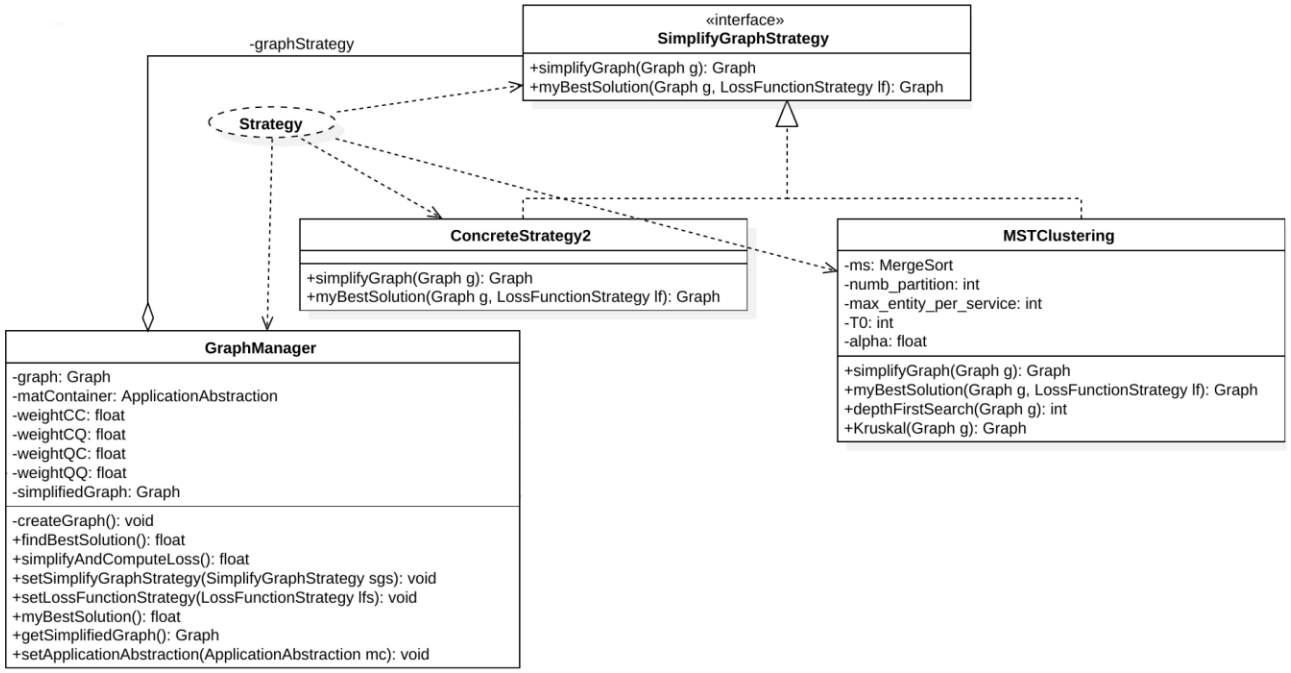


Semplificazione

- La fase di semplificazione prevede l'uso della maggior parte delle classi descritte nella slide precedente:
  - **SimplifyGraphStrategy:** rappresenta uno strategy pattern
  - **LossFunctionStrategy**: rappresenta uno strategy pattern
  - **StandardSGF**: rappresenta una classe factory di un AbstractFactoryPattern
  - **LossFunctionFactory:** rappresenta un factory pattern

SimplifyGraphStrategy

- Siccome potrebbero esserci metodi di semplificazione di grafi diversi da quello realizzato attualmente, è stato proposto un design pattern Strategy in modo da ampliare questa funzionalità a nuovi algoritmi.
- L'interfaccia presenta 2 metodi:
  - **SimplifyGraph(Graph g)**: permette di semplificare il grafo con iperparametri fissi (ad esempio, nel caso del MSTClustering gli iperparametri sono il numero max di entità per microservizio e il numero di microservizi)
  - **MyBestSolution(Graph g, LossFunctionStrategy lf)**: Algoritmo di ricerca che prova a trovare una soluzione migliore possibile cambiando gli iperparametri dell'algoritmo.



Semplificazione del grafo

Alle funzioni dello strategy si accede tramite le corrispettive funzioni di **GraphManager**: **myBestSolution**(..) e **simplifyAndComputeLoss**(..).

Entrambe le due funzioni ritornano il valore della loss function associato al grafo semplificato, mentre il grafo viene salvato in un attributo di GraphManager **simplifiedGraph**, a cui si può accedere tramite il metodo **getSimplifiedGraph**().

Semplificazione del grafo

In **GraphManager** è presente anche un altro metodo:

- **findBestSolution**: Metodo che esegue l'algoritmo **myBestSolution**, per ogni strategia di semplificazione implementata nel progetto. Viene scelta poi la soluzione che risulta con la minore perdita tra tutte.
  Quindi, scelta una funzione di Loss, il risultato sarà la miglior semplificazione tra tutte.

Questo metodo è stato aggiunto anche per dare una stima veloce di come sarebbe la semplificazione tra le varie strategie implementate.

Utilizzo del factory pattern **StandardSGF.**

MSTClustering

Classe che implementa dei metodi concreti per la semplificazione di un grafo pesato.

MSTClustering
-ms: MergeSort -numb_partition: int -max_entity_per_service: int -T0: int -alpha: float
+simplifyGraph(Graph g): Graph +myBestSolution(Graph g, LossFunctionStrategy lf): Graph +depthFirstSearch(Graph g): int +Kruskal(Graph g): Graph

MSTClustering

La classe implementa nel metodo **simplifyGraph** l'algoritmo di**[1]**:

- Esecuzione algoritmo di kruskal per trovare un MST
- Ordinamento in modo decrescente sul peso dei nodi
- Eliminazione degli archi finché non si arriva ad una soglia di partizioni ottenute
- Controllo sul numero di vertici per partizione
- Si reinseriscono poi gli archi tagliati dall'algoritmo di kruskal dei nodi che si ritrovano nelle stesse partizioni.

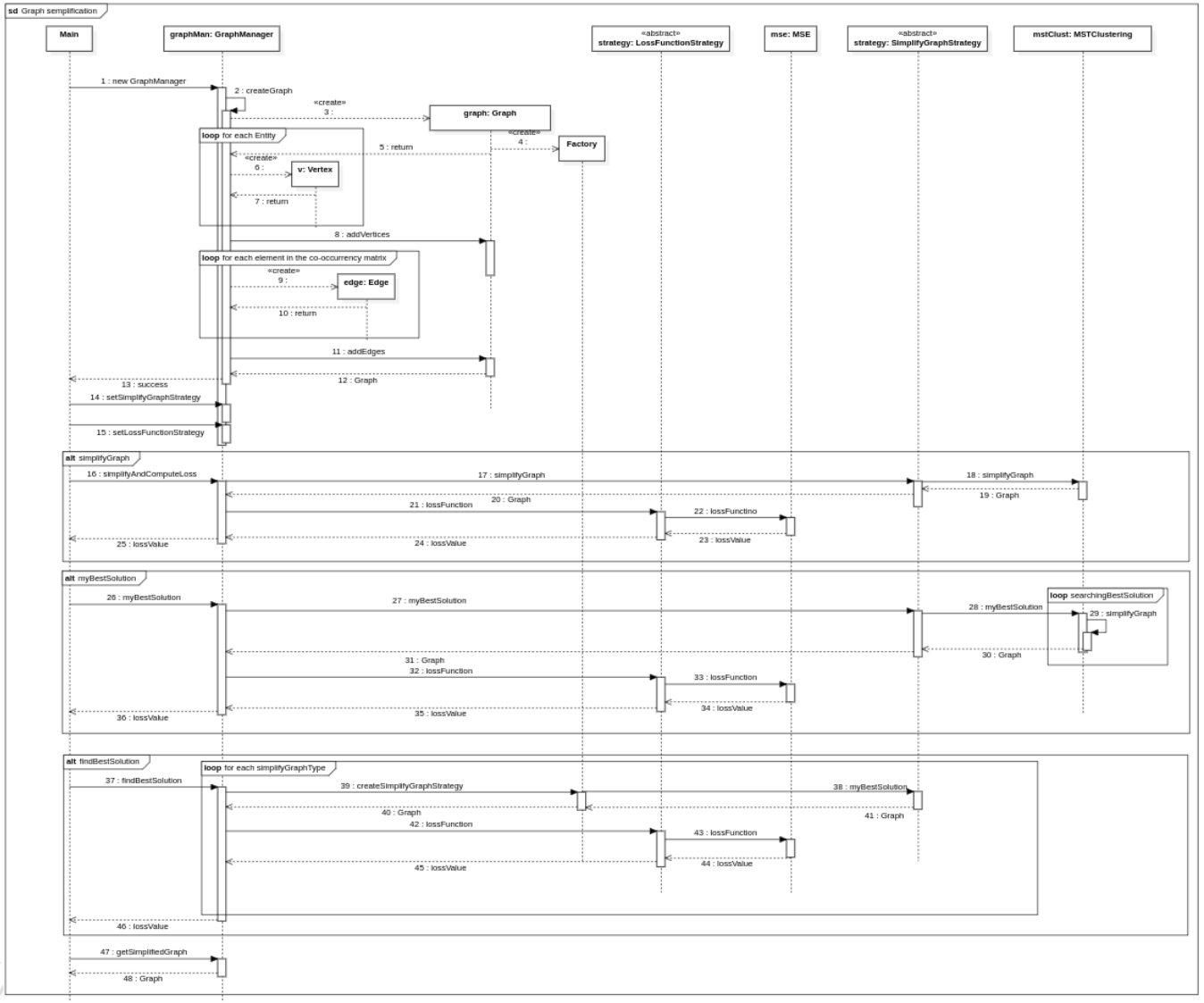
MSTClustering

Il metodo **myBestSolution**:

- Implementa un algoritmo di ricerca ispirato al **Simulated Annealing**:
  - Iterando su varie combinazioni di numero di partizioni e sul numero di entità per partizione.
  - L'algoritmo nelle prime iterazioni accetta di andare anche verso soluzioni svantaggiose. Man mano che si avanza con le iterazioni, si diminuisce la probabilità di accettare questo tipo di soluzioni fino ad azzerarla.

Essendo un algoritmo di ricerca locale, non è detto che la soluzione migliore trovata sia quella migliore assoluta.

Funzionamento Algoritmi

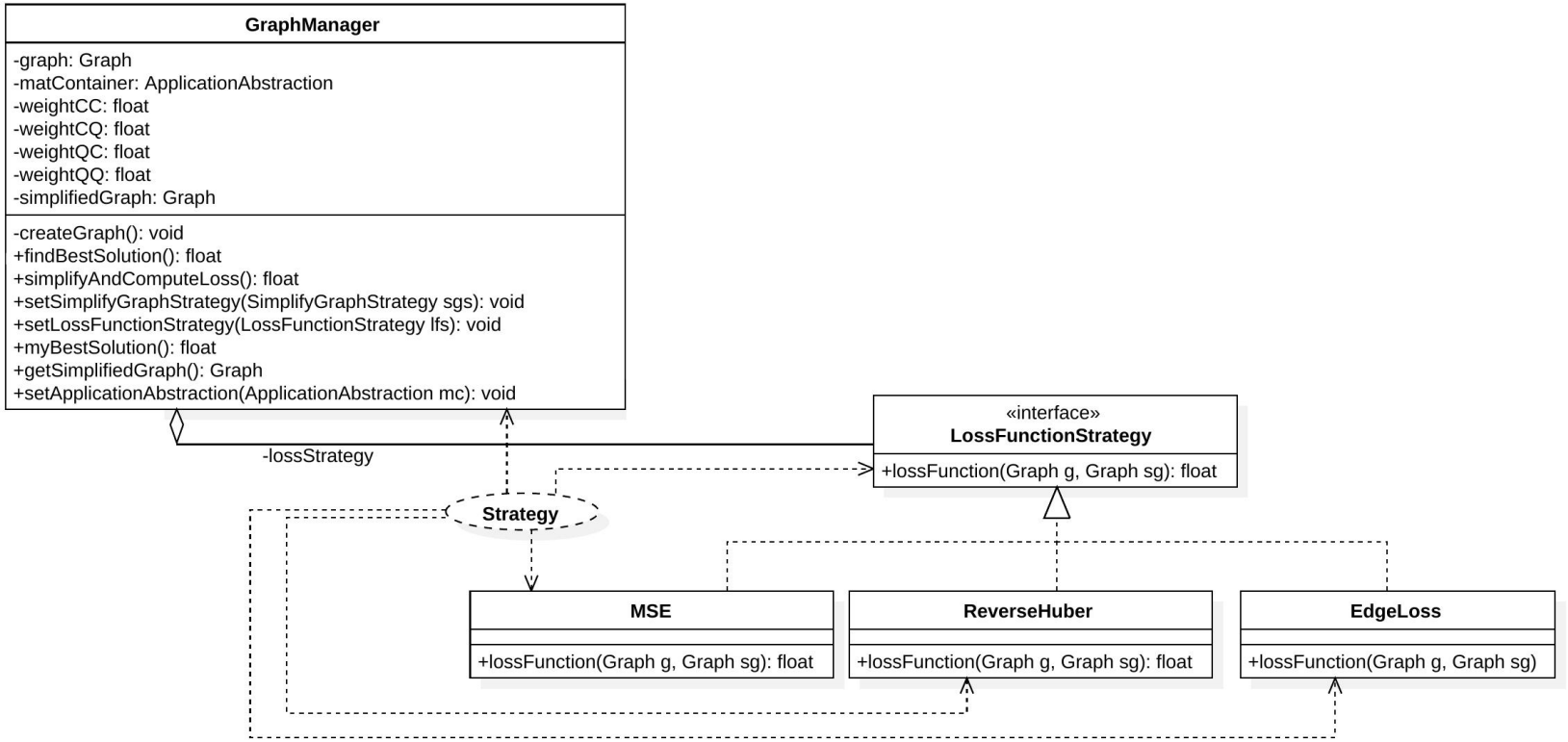


Tecniche di Loss

- Le funzioni di loss vengono normalmente usate per valutare modelli predittivi.
- In questo caso le funzioni di loss:
  - Servono a valutare quanto la soluzione è migliore rispetto ad altre o rispetto all'applicazione monolitica di partenza.
  - Lavorano sui pesi degli archi del grafo.

È stato deciso di aggiungere uno **strategy pattern** per garantire la possibilità di implementare diverse loss function.

Tecniche di Loss



Tecniche di loss

- Sono state implementate 3 loss function che funzionano in modo molto diverso:
  - **MSE**: Si implementa una specie di Mean Square Error calcolata sui pesi degli archi mancanti dalla soluzione semplificata.
  - **Reverse Hubert**: Questa funzione di loss penalizza maggiormente gli errori grandi rispetto a quelli piccoli, utilizzando il quadrato del peso per gli errori grandi e il peso stesso per gli errori piccoli:
    - $Formula: \begin{cases} wt & se\ wt \leq th \\ \frac{wt^2+th^2}{2th} & altrimenti \end{cases}$
      - wt: peso di un arco, in percentuale (quindi moltiplicato per 100)
      - th: threshold, $th = 0.8\left(max_{i \in eL}(wt_i)\right)$, eL = insieme degli archi tagliati.
  - **EdgeLoss**: Questa tecnica effettua una valutazione della loss basandosi sul numero di componenti connesse e sulla forza del legame tra entità appartenenti ad una componente connessa.

EdgeLoss

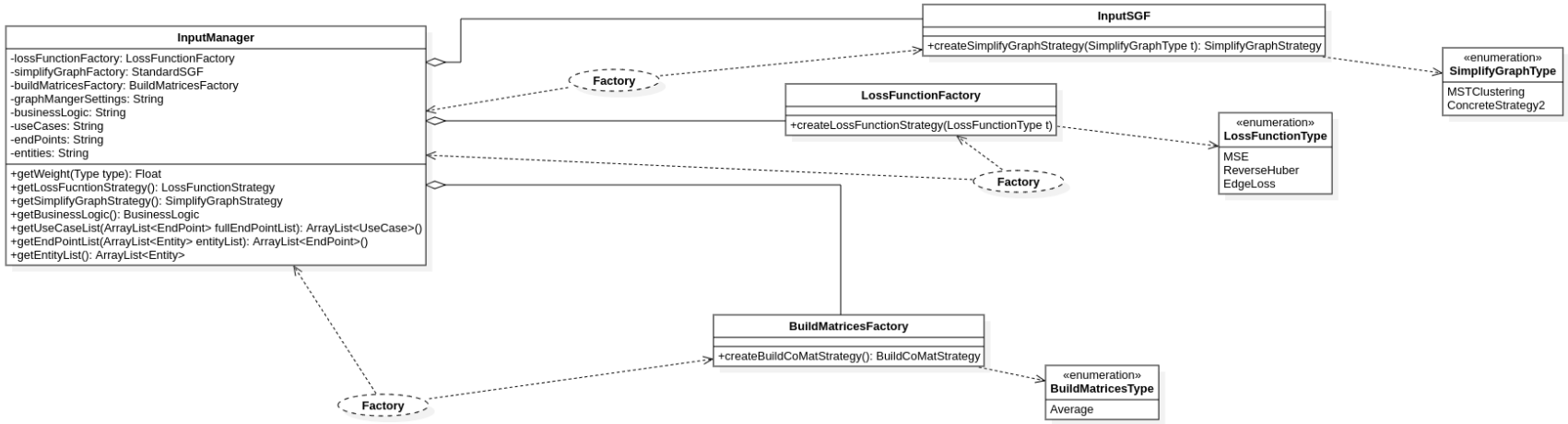
- La tecnica EdgeLoss valuta la loss tenendo conto di due fattori:
  1. Numero di componenti connesse generate dalla tecnica di semplificazione. In particolare:
     - Si effettua una stima del numero di componenti connesse desiderate per un determinato numero di entità:
       - des = $stima(x) = \frac{x}{log_2 x}$, con x il numero di entità.
       - Si aggiunge alla loss: $\begin{cases} e^{2*\frac{numCC}{des}} & se\ numCC < des \\ e^{2*\frac{des}{numCC}} & altrimenti \end{cases}$
         - » $numCC$ è il numero di componenti connesse effettivo.
  2. Media degli archi contenuti all'interno di una stessa componente connessa. Se la media dei pesi degli archi relativi ad una specifica componente connessa è inferiore ad una certa soglia, si aumenta il valore della loss in maniera più sostanziosa.

Input

- È stata realizzata InputManager:
  - La classe inserisce la funzionalità di leggere da file json i vari dati in ingresso descritti precedentemente.

InputManager
-lossFunctionFactory: LossFunctionFactory -simplifyGraphFactory: SimplifyGraphFactory -buildMatricesFactory: BuildMatricesFactory -graphMangerSettings: String -businessLogic: String -useCases: String -endPoints: String -entities: String
+getWeight(Type type): Float +getLossFucntionStrategy(): LossFunctionStrategy +getSimplifyGraphStrategy(): SimplifyGraphStrategy +getBusinessLogic(): BusinessLogic +getUseCaseList(ArrayList<EndPoint> fullEndPointList): ArrayList<UseCase>() +getEndPointList(ArrayList<Entity> entityList): ArrayList<EndPoint>() +getEntityList(): ArrayList<Entity>

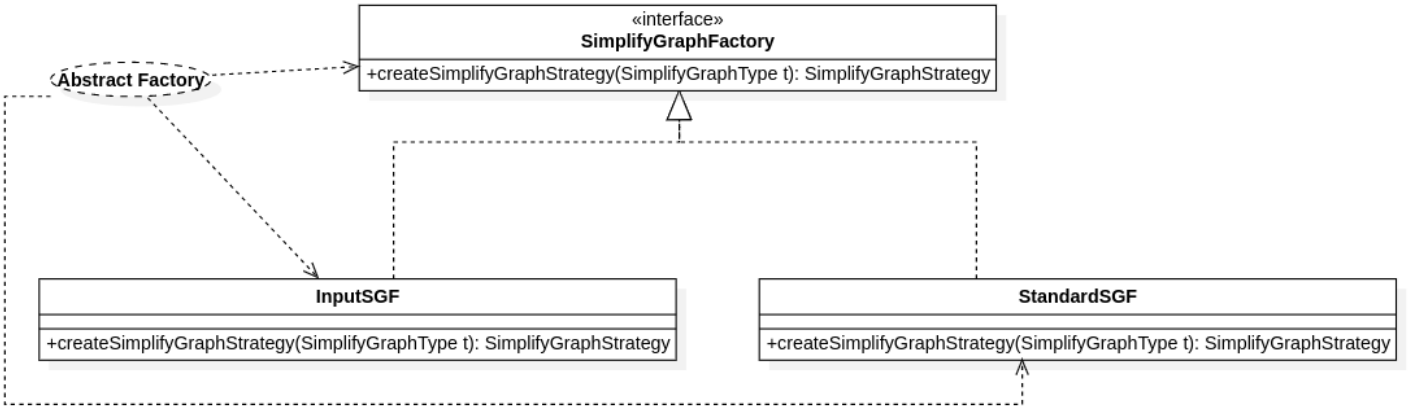
Input



Sono state realizzate delle **factory** per ottimizzare la creazione delle strategie di costruzione delle matrici di co-occorrenza (**BuildMatricesFactory**) e delle strategie di calcolo della loss (**SimplifyGraphFactory** e **LossFunctionFactory**), a partire dall'input dei file JSON.

AbstractFactory

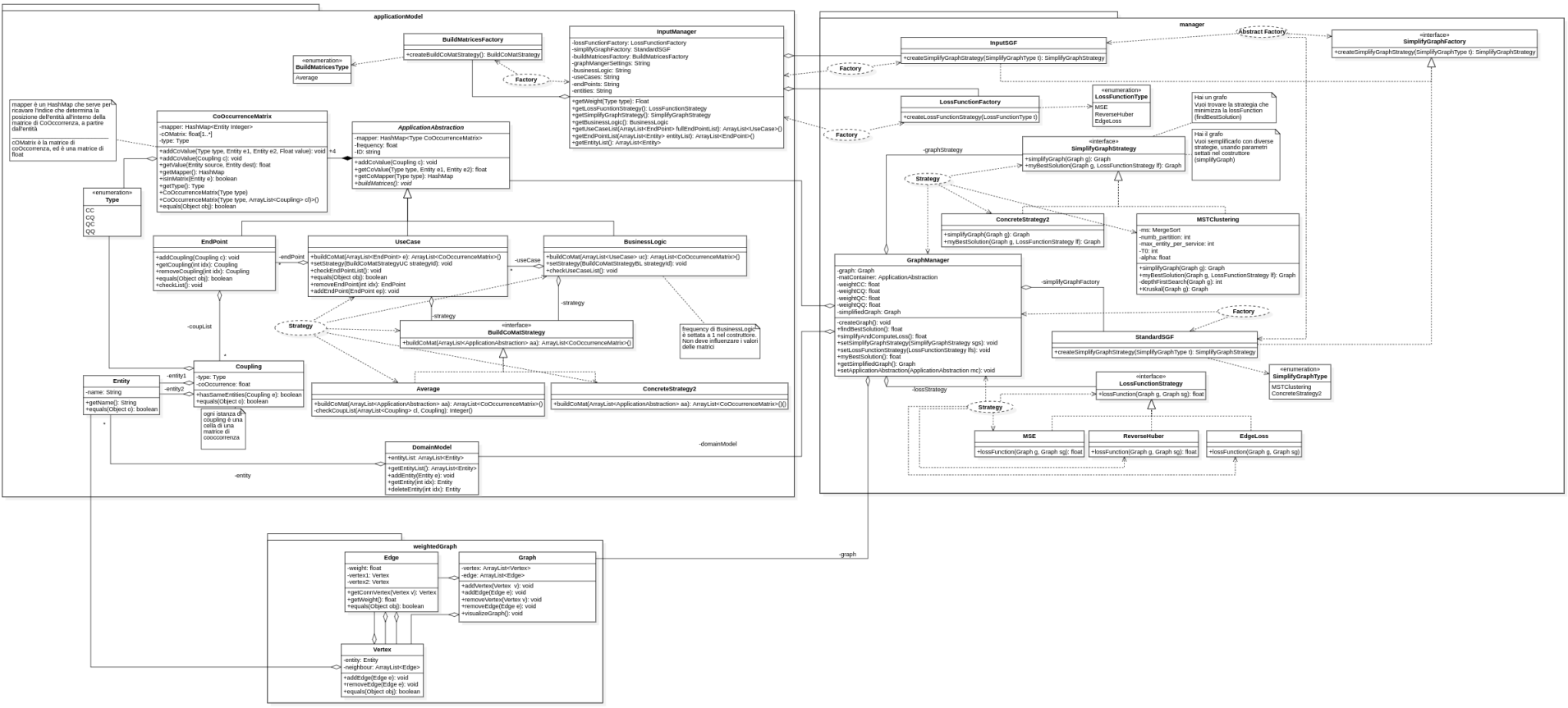
- È stato deciso di inserire un **Abstract Factory pattern** per simplifyGraphStrategy:



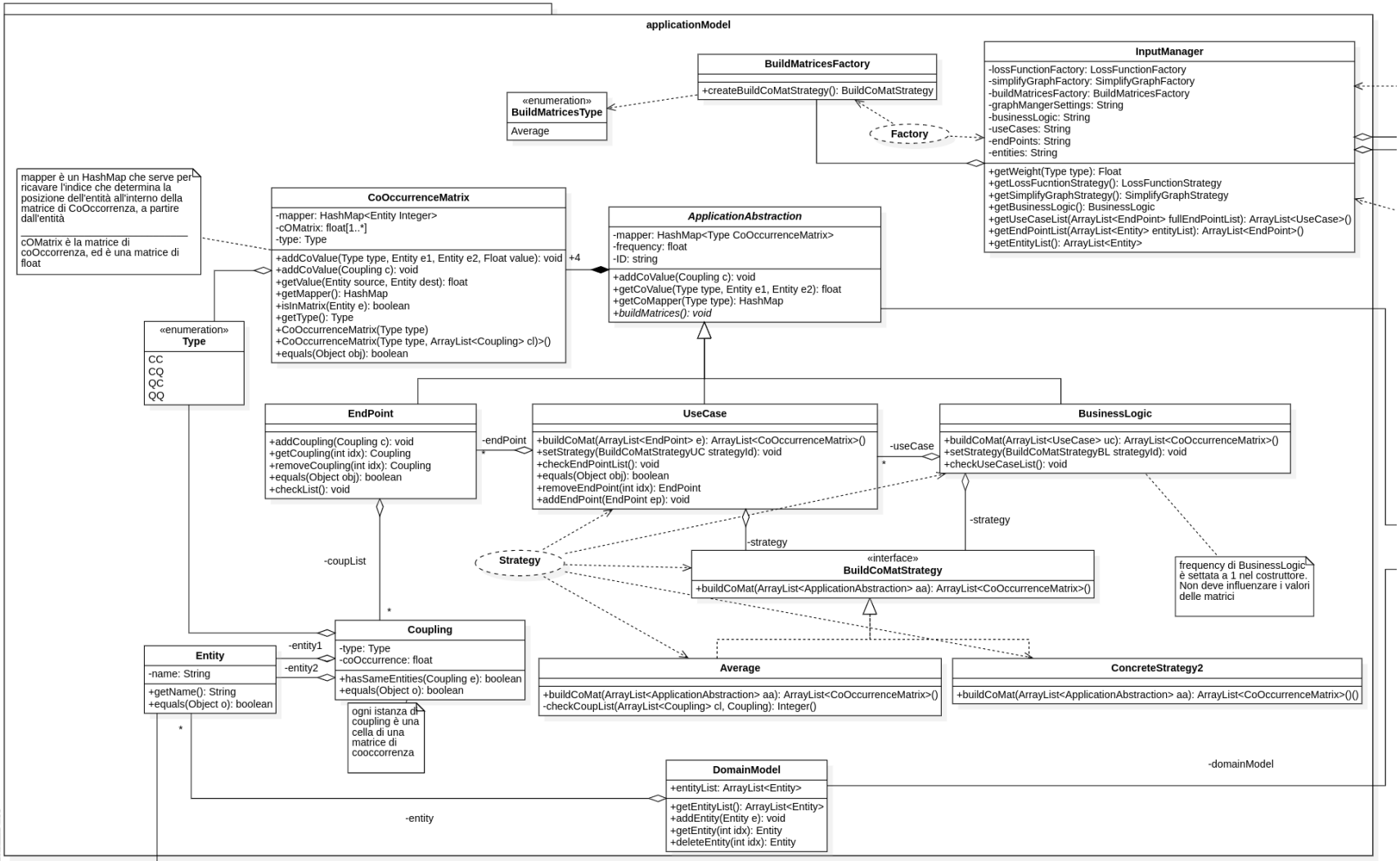
- In questo modo si riesce ad avere un'implementazione diversa per l'input in cui si può richiedere a runtime i parametri della strategia direttamente dentro il factory. Così riusciamo a costruire l'oggetto di tipo **SimplifyGraphStrategy** con i parametri che l'utente desidera.
- Questo porta ad avere la classe factory fortemente accoppiata alle operazioni di input. D'altro canto ci permette di prendere in input parametri diversi a seconda della strategia.

Refactoring

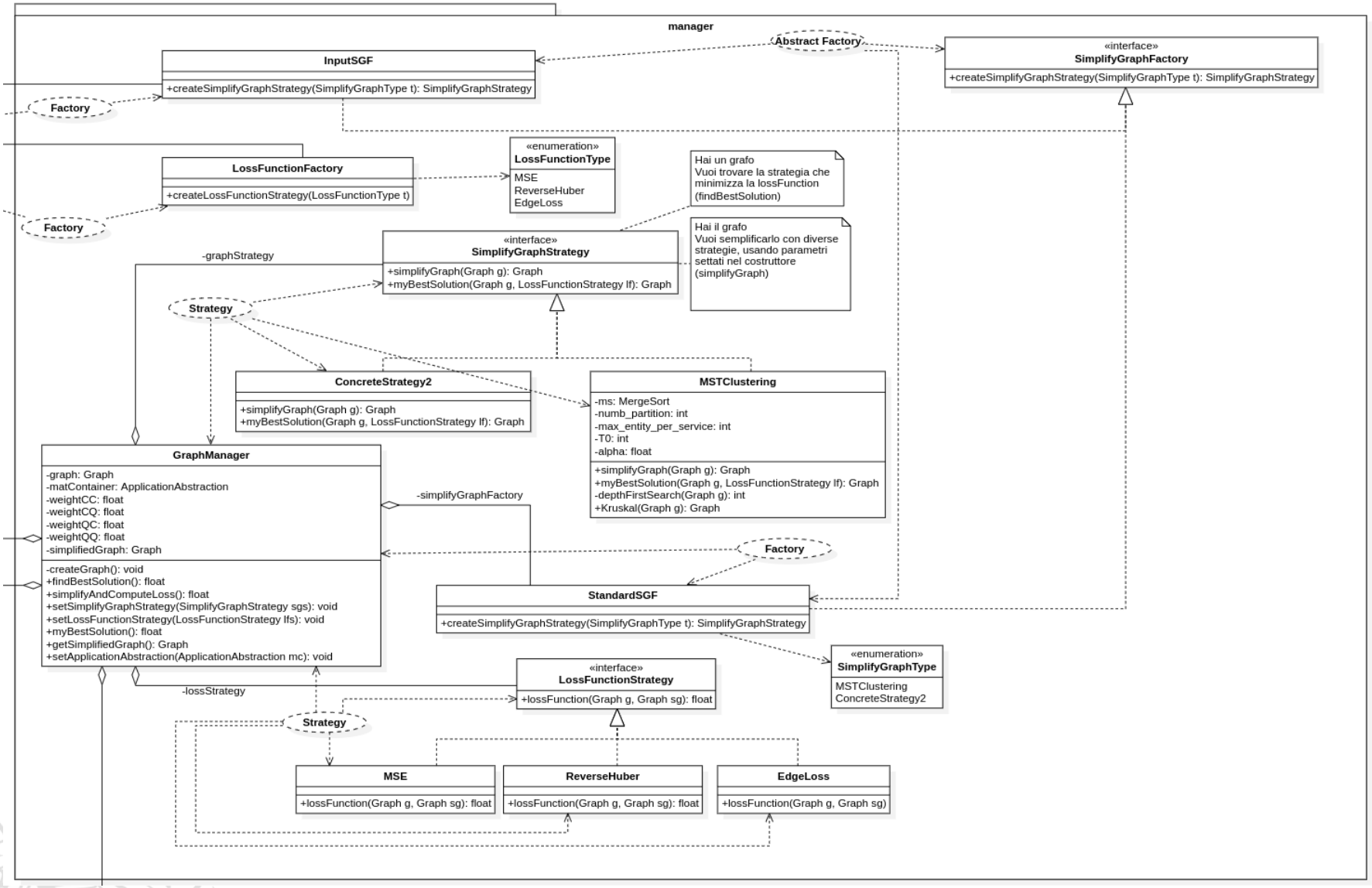
- Di seguito il class diagram di tutto il progetto:



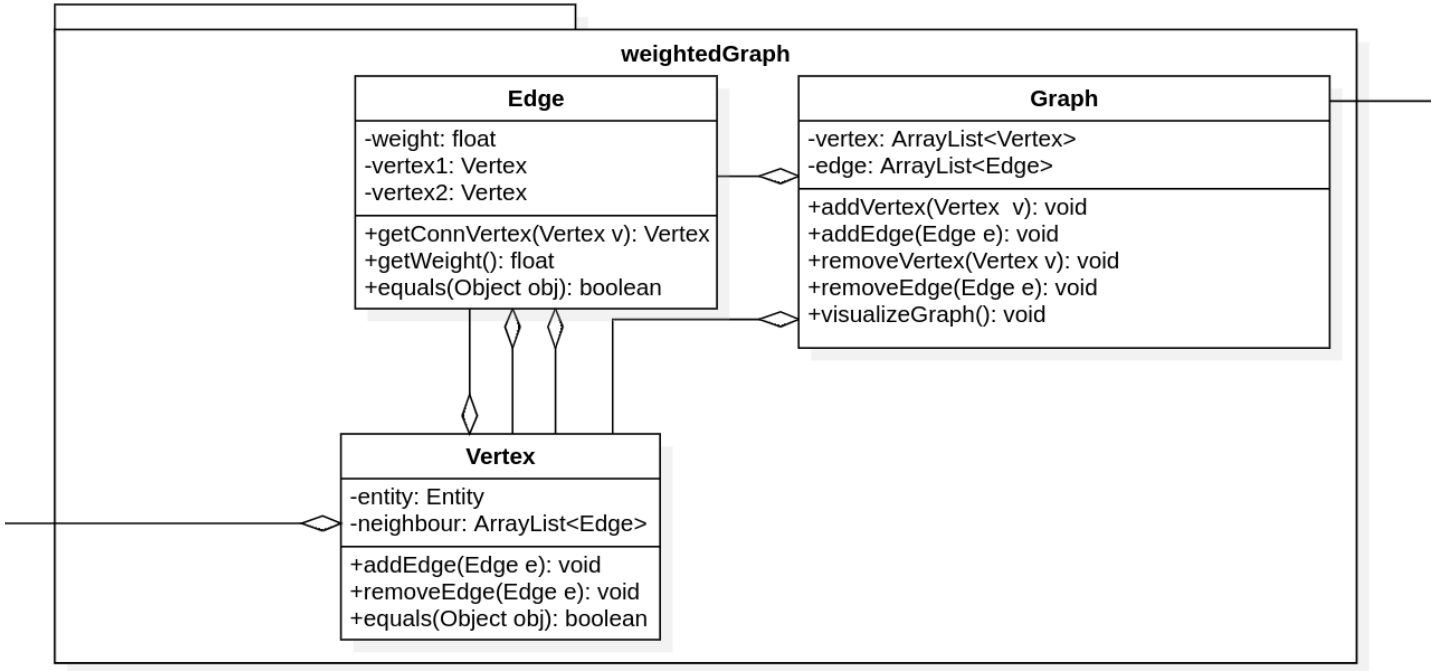
Application Model



Manager


manager
Abstract Factory
InputSGF
+createSimplifyGraphStrategy(SimplifyGraphType t): SimplifyGraphStrategy
«interface»
SimplifyGraphFactory
+createSimplifyGraphStrategy(SimplifyGraphType t): SimplifyGraphStrategy
Factory
LossFunctionFactory
+createLossFunctionStrategy(LossFunctionType t)
«enumeration»
LossFunctionType
MSE
ReverseHuber
EdgeLoss
Factory
Hai un grafo
Vuoi trovare la strategia che minimizza la lossFunction (findBestSolution)
Hai il grafo
Vuoi semplificarlo con diverse strategie, usando parametri settati nel costruttore (simplifyGraph)
-graphStrategy
«interface»
SimplifyGraphStrategy
+simplifyGraph(Graph g): Graph
+myBestSolution(Graph g, LossFunctionStrategy lf): Graph
Strategy
ConcreteStrategy2
+simplifyGraph(Graph g): Graph
+myBestSolution(Graph g, LossFunctionStrategy lf): Graph
MSTClustering
-ms: MergeSort
-numb_partition: int
-max_entity_per_service: int
-T0: int
-alpha: float
+simplifyGraph(Graph g): Graph
+myBestSolution(Graph g, LossFunctionStrategy lf): Graph
-depthFirstSearch(Graph g): int
+Kruskal(Graph g): Graph
GraphManager
-graph: Graph
-matContainer: ApplicationAbstraction
-weightCC: float
-weightCQ: float
-weightQC: float
-weightQQ: float
-simplifiedGraph: Graph
-createGraph(): void
+findBestSolution(): float
+simplifyAndComputeLoss(): float
+setSimplifyGraphStrategy(SimplifyGraphStrategy sgs): void
+setLossFunctionStrategy(LossFunctionStrategy lfs): void
+myBestSolution(): float
+getSimplifiedGraph(): Graph
+setApplicationAbstraction(ApplicationAbstraction mc): void
-simplifyGraphFactory
Factory
StandardSGF
+createSimplifyGraphStrategy(SimplifyGraphType t): SimplifyGraphStrategy
«enumeration»
SimplifyGraphType
MSTClustering
ConcreteStrategy2
-lossStrategy
«interface»
LossFunctionStrategy
+lossFunction(Graph g, Graph sg): float
Strategy
MSE
+lossFunction(Graph g, Graph sg): float
ReverseHuber
+lossFunction(Graph g, Graph sg): float
EdgeLoss
+lossFunction(Graph g, Graph sg)

Weighted Graph



References

- **[1]** Mazlami, G., Cito, J., & Leitner, P. (2017, June). Extraction of microservices from monolithic software architectures. In *2017 IEEE International Conference on Web Services (ICWS)* (pp. 524-531). IEEE.