Alma Mater Studiorum – Università
di Bologna

Corso di Laurea Magistrale in Scienze e
Tecnologie Informatiche

Progetto Sistemi Distribuiti

Users in MoK: Epistemic Actions for Knowledge Formation and Discovery

Nicola Casadei

nicola.casadei4@studio.unibo.it

Indice

Capitolo 1

Introduzione

1.1 Introduzione

Questo è il report di progetto del corso di Sistemi Distribuiti dell'Università di Bologna. Di seguito sarà consultabile il processo di analisi del progetto, con un aumento incrementale del dettaglio del dominio e delle interazione delle entità appartenenti ad esso. La relazione inizia con l'analisi e la modellazione del concetto di Azione, il modello verrà poi ampliato con le altre entità del Dominio. Infine verrà presentata e implementata una possibile architettura per la gestione delle Perturbation action.

Per qualsiasi dubbio in merito fare riferimento all'autore.

1.2 Visione

Il progetto si propone come una possibile rappresentazione delle *Action* eseguibili all'interno di un sistema distribuito *MoK*.
Molecule of Knowledge(*Mok*) nasce con l'obiettivo di offrire un modello di Coordinazione in un sistema Socio-Tecnico (*STS*) ad alto contenuto informativo (*KIE*) concentrandosi in particolare sugli ~~aspetti di: *Self-Organising* e *User-Centric*.~~
In questo specifico contesto, la rappresentazione delle azioni è una parte fondamentale del sistema ~~MoK. Poiché~~ rappresenta la principale modalità di interazione dell'utente con il sistema.

Il progetto, inoltre, vuole offrire una possibile architettura per la gestione delle *Perturbazioni* che l'esecuzione di un'azione ha sul sistema. In particolare si vuole concentrare sul ricavare informazioni dal contesto in cui è stata eseguita l'azione e a sua volta generare una perturbazione coerente.

1.3 Obiettivi

Lo scopo del progetto è quello di individuare un possibile insieme di *Action* eseguibili da un utente del sistema MoK, rappresentarle tramite un modello e utilizzare lo stesso per l'implementazione di una piccola porzione di Sistema MoK. Il secondo obiettivo dell'elaborato è progettare un'architettura capace di intercettare l'esecuzione delle *Action* e, tramite l'analisi dello stato del sistema (o di una sua porzione), eseguire una azione di perturbazione, che indicheremo come *Perturbation*.
Similmente alle Action, anche le Perturbation necessiteranno di un modello che verrà poi implementato per qualche semplice Test.

Capitolo 2

Action

2.1 Action

Un sistema MoK è un particolare tipo di sistema socio-tecnico, il quale necessita dell'interazione dell'utente per funzionare: di conseguenza le azioni sono il principale metodo con cui l'utente può interagire sul sistema.
Il primo passo fondamentale prima della rappresentazione delle azioni é individuare una lista di Azioni che giá sono utilizzate nell'ambito di STS particolarmente famosi, come ad esempio: *Facebook*, *Twitter* e *Linkedin*. L'analisi di questi STS, integrata con quella giá effettuata dal Dottor Stefano Mariani [1], ha permesso di individuare un insieme di possibili *Action*.

2.1.1 L'insieme di *Action*

In questa sezione si riporta l'elenco delle azioni individuate dall'analisi, scelte sulla base delle seguenti motivazioni:

- L'azione è presente, anche se con nomi diversi, in tutti i sistemi socio-tecnici analizzati;
- l'azione è parte fondamentale dell'interazione dell'utente con il sistema.

Le azioni scelte sono:

- *share*: l'azione permette di creare o diffondere nuove informazioni nel sistema;
- *mark*: l'azione permette di evidenziare, votare o salvare informazioni apprezzate o considerate utili dall'utente;
- *annotate*: l'azione permette di commentare o più in generale accodare nuove informazioni ad informazioni giá esistenti;
- *connect*: l'azione permette di creare un collegamento, relazione con un altro agente del sistema;
- *harvest*: l'azione permette un qualsiasi tipo di ricerca all'interno del sistema.

Una volta individuato questo gruppo di azioni si è passato alla rappresentazione di un modello per la rappresentazione delle azioni stesse e del dominio in cui esse dovranno esistere ed essere eseguite.

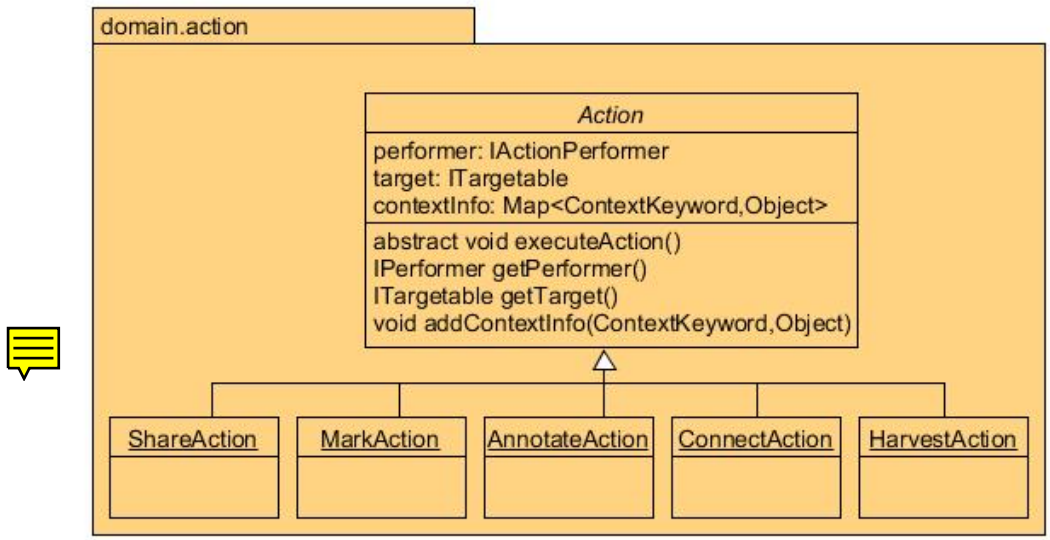

Figura 2.1: Modello delle *Action*

Come mostrato nel modello, ogni *Action* è una classe Astratta, le singole azioni eseguibili sul sistema specializzano la classe *Action* implementando, in modo coerente agli scopi del sistema, la procedura *executeAction()*. La classe *Action* ha tre componenti comuni ad ogni azione:

1. **performer**: è l'agente che ha eseguito l'*Action*, nel modello gli agenti che hanno il potere di eseguire le Action sono di tipo *IPerformer*;

2. **target**: è la parte del sistema che subisce l'azione, può essere un *Atom*, *Molecule*, *Seed*, *User* o nei particolari casi di *HarvestAction* può essere un *Compartment* o il sistema stesso;

3. **contextInfo**: è una mappa di valori in cui sono memorizzate le informazioni che descrivono il contesto in cui è stata eseguita l'azione, possono essere utilizzate per generare gli *Enzym* e le *Trace* più adatte al particolare stato del sistema o di una sua porzione; per questo motivo è presente la funzione *addContextInfo* per aggiungere le informazioni necessarie all'analisi di contesto.

2.1.2 Le informazioni di contesto

L'idea che sta alla base di questa implementazione è la grande eterogeneità fra le possibili informazioni che possono essere utili ad un qualsiasi sistema MoK per descrivere in dettaglio il contesto di esecuzione di una particolare *Action*.
La coppia keyword - Object può individuare un qualsiasi tipo di informazione che può essere poi utilizzata correttamente tramite un Cast opportuno.
Seppure la definizione di *ContextKeyword* e l'analisi del contesto esuli dagli obiettivi del progetto, a scopo informativo verrà di seguito presentato un insieme di informazioni che possono risultare utili per la descrizione del contesto esecutivo:

- **previousAction - Action**: questa coppia chiave valore identifica l'azione che ha preceduto l'azione temporale immediatamente precedente a quella di interesse o l'azione precedente eseguita dallo stesso User;

- **followingAction - Action**: il concetto è simile a quello precedente ma si riferisce ad un'azione compiuta dopo quella di interesse;

- **time - Datetime**: rappresenta l'istante temporale in cui è stata notificata l'esecuzione dell'azione

Si noti come le informazioni di contesto di interesse dipendano anche dagli scopi del Sistema: Social-Network, diffusione di articoli scientifici e molti altri.

Capitolo 3

Domain Model

3.1 Modello del Dominio

Una volta definito il modello delle *Action*, il modello del Dominio è completato in modo tale che sia coerente con le scelte fatte per la rappresentazione scelta.

3.1.1 Performer e Target

Il modello delle azioni introduce il concetto di *Performer* e *Target*, che rappresentano rispettivamente chi esegue e chi subisce l'azione. I due concetti sono stati formalizzati tramite l'implementazione delle due Interface:*IPerformer* e *ITargetable*; sia il Performer che il Target condividono il concetto di ***correlations***:

- nel caso del Target, tale concetto rappresenta a quali "temi" le informazioni che esso contiene sono più attinenti;
- per il Performer le correlations rappresentano gli interessi dell'entità a particolari informazioni.

In generale, il concetto di *correlations* può risultare utile per valutare la similarità fra due o più Atomi o Molecole del sistema. Il concetto di similarità tornerà utile più avanti in particolare per l'esecuzione delle *Perturbation*.

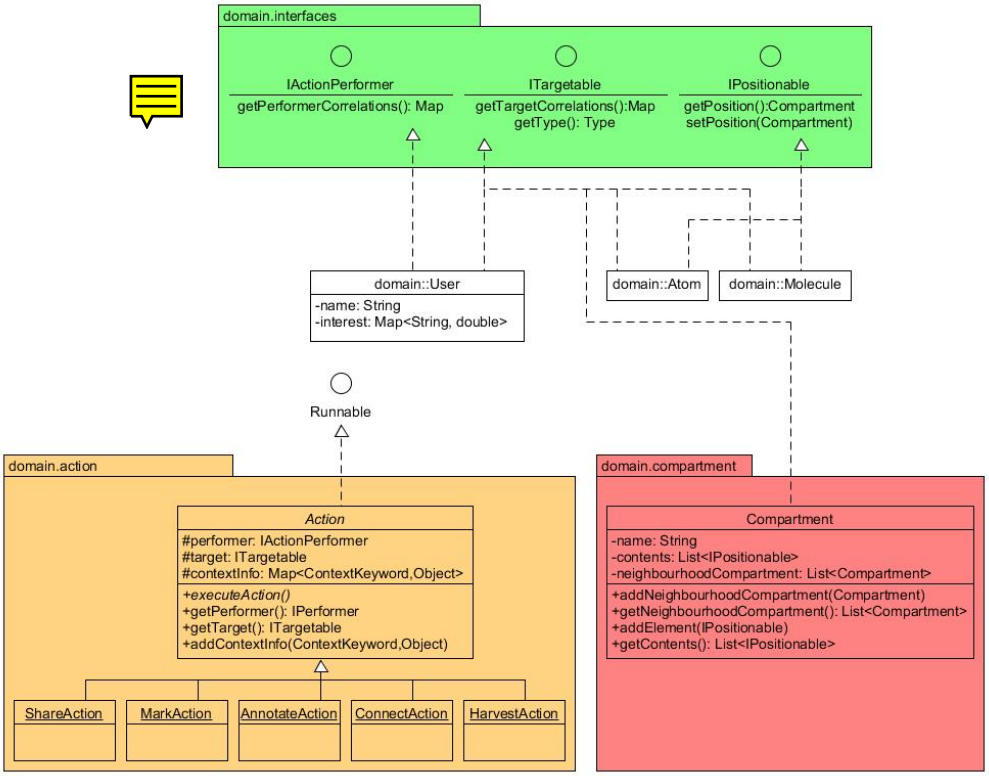

Figura 3.1: Modello del *Dominio*

Lato Performer, le *correlations* o *interests* possono risultare utili per proporre all'agente contenuti in linea coi suoi interessi e migliorare la sua esperienza col sistema. Quest'ultima caratteristica seppur non implementata è stata aggiunta per completezza.

3.1.2 Locazione e Compartment

L'interazione con un qualsiasi elemento MoK è possibile solo se questo è logicamente localizzabile all'interno del sistema; per questo motivo è stato implementato il concetto MoK di *Compartment*, un contenitore logico di elementi del sistema considerati *posizionabili*, proprietà da cui nasce l'Interface *IPositionable*.
Il *Compartment* è formato da due componenti fondamentali:

- **contents**: questa lista rappresenta tutte le entità che sono logicamente contenute nel Compartment, queste entità sono generalizzate tramite l'Interace *IPositionable*;
- **neighbourhoodCompartment**: è una lista dei Compartiment logicamente vicini all'entità.

Il concetto di Compartment non solo è utile a fine organizzativo allo scopo di distribuire gli elementi sul sistema, ma risulta fondamentale per la successiva implementazione delle *Perturbation*; infatti si rivela importante poter localizzare la posizione in cui un'azione ha avuto luogo.

Capitolo 4

Enzyme e Trace

4.1 Enzimi e Tracce

Una volta definiti gli elementi che saranno in qualche modo parte fondamentale dell'azione, siano essi i Performer o i Target della stessa, è necessario definire delle entità del Dominio che permettano al sistema di analizzare l'azione appena eseguita e il contesto in cui è stata eseguita. Da questa analisi sarà poi possibile eseguire delle opportune azione implicite già definite come *Perturbation*.

Come si può vedere dal modello del Dominio, sono state aggiunte due Entità di tipo *IPositionable*: *Enzyme* e *Trace*

4.1.1 Enzyme

Gli Enzyme rappresentano in qualche modo l' "eco" che l'azione lascia nel Sistema, contiene quindi tutte le informazioni necessarie a descrivere l'Azione eseguita e il contesto in cui è stata eseguita.

4.1.1.1 Attributi

- **species**: rappresenta il tipo di azione eseguita. Il valore quindi corrisponde con un elemento dell'insieme delle azioni implementate: Share, Mark, Annotate, Connect o Harvest.

- **target**: rappresenta il bersaglio che ha subito l'azione, questa è un'in-

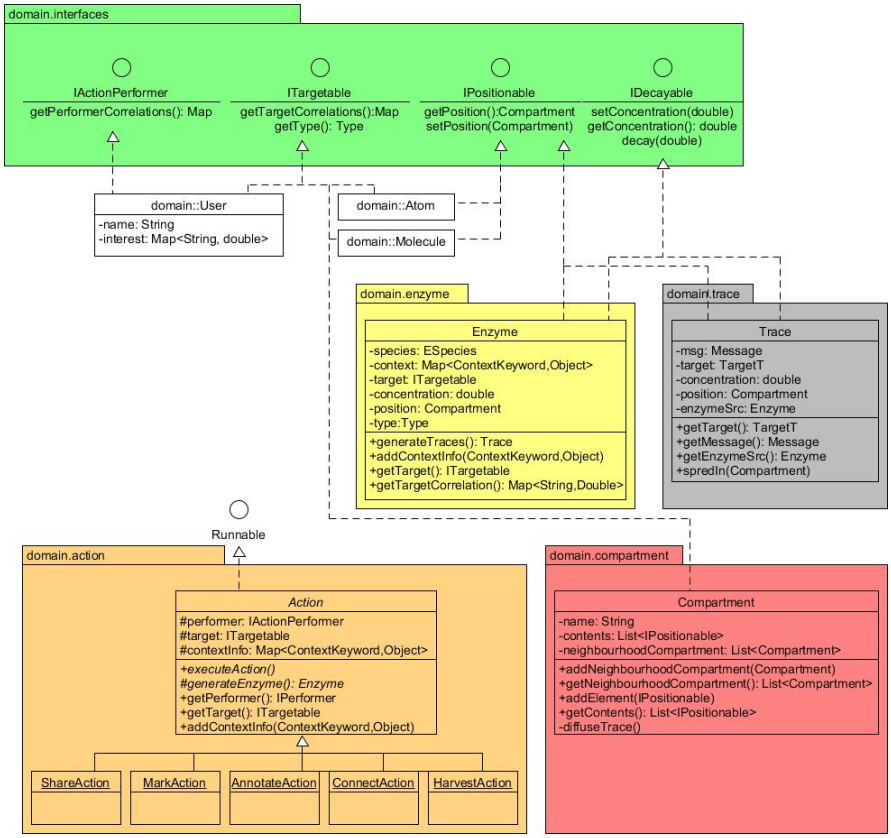

Figura 4.1: Modello del *Dominio* con l'aggiunta di Enzimi e Tracce

formazione fondamentale poichè quasi tutte le azioni di *Perturbation* si basano sulla similarità fra diverse entità del sistema, in particolare Atomi e Molecole;

- **position**: essendo l'Enzyme una entità IPositionable, può essere contenuta all'interno dei *Compartment*;

- **ContextInfo**: questa mappa, similmente all'entità Action, conterrà tutte le informazioni aggiuntive che possano aiutare a descrivere con maggiore dettaglio il contesto in cui l'azione, e quindi l'Enzyme, è stato generato.

4.1.1.2 Funzioni fondamentali

- *generateTrace*: questa funzione è il cuore dell'*Enzyme*, genera in base all'analisi del contesto una traccia coerente ad esso.

4.1.2 Trace

Le Tracce o *Trace* sono entità generate dall'Enzyme in base al contesto da esso analizzato, e costituiscono il passo che precede l'esecuzione di una azione implicita o *Perturbation*. Con questa implementazione è possibile che uno stesso enzima possa generare, in base al contesto, Trace quindi *Perturbation* diverse.

4.1.2.1 attributi

- **msg**: rappresenta il messaggio implicito scaturito dall'analisi del contesto esecutivo e dello stato del sistema, il messaggio determinerà il tipo di *Perturbation* da eseguire;

- ~~**target**: a differenze dell'Enzyme non rappresenta una particolare entità, ma la famiglia di Entità che devono essere bersaglio delle azioni di *Perturbation*;~~

- **position**: anche le *Trace* come l'Enzyme è contenuta all'interno di un Compartment;

- **enzymeSrc**: questo attributo rappresenta un riferimento all'enzima generante nel caso in cui le informazioni da esso contenute risultino rilevanti alla *Trace*.

Le tracce possono diffondersi nei compartimenti vicini, per farlo è stata implementata la funzione: *spreadIn(compartment, rate)*: ~~la quale crea nel compartment una copia della *Trace* con un valore di concentrazione pari a *rate*. Lo stesso valore *rate* verrà sottratto alla concentration della Trace che ha chiamato la funzione.~~

4.1.3 Le entità Decayable

~~A differenza di elementi persistenti come *Atom*, *Molecule*, *Compartment*,~~ etc., le entità *Enzyme* e *Trace* precedentemente introdotte hanno natura

temporanea all'interno del sistema. Per la completezza del modello, risulta quindi necessario presentare un' opportuna interfaccia capace di rappresentare la decadibilità di queste entità. Tale interfaccia viene definita *IDecayable*:

- **concentration**: rappresenta la concentrazione residua dell'entità all'interno di un *Compartment*, quando essa raggiungerà lo zero l'entità sarà considerata completamente decaduta e ogni suo riferimento potrà essere eliminato dal sistema.
- funzioni di supporto:
  - **setConcentration**: con questa funzione è possibile modificare il valore concentration di una entità in base a particolari necessità del sistema;
  - **decay(rate)**: la funzione riduce di un valore *rate* la concentrazione dell'entità

4.1.4 L'interazione delle Entità

Il modello di interazione presentato nello schema rappresenta il comportamento generale del Sistema quando viene eseguita un'azione. Si noti come l'*Action* si occupa di generare l'*Enzyme*, il quale a sua volta analizza il contesto e genera una *Trace* opportuna, infine entrambe le nuove entità generate vengono aggiunte ad un *Compartment* opportuno. Nel progetto si ipotizza che tale *Compartment* sia quello che contiene il bersaglio della *Action*.
Seppure non implementato, per completezza è stato aggiunto al diagramma l'interazione dell'Utente con una opportuna interfaccia grafica dalla quale è possibile eseguire le azioni e visualizzarne gli effetti sul sistema.

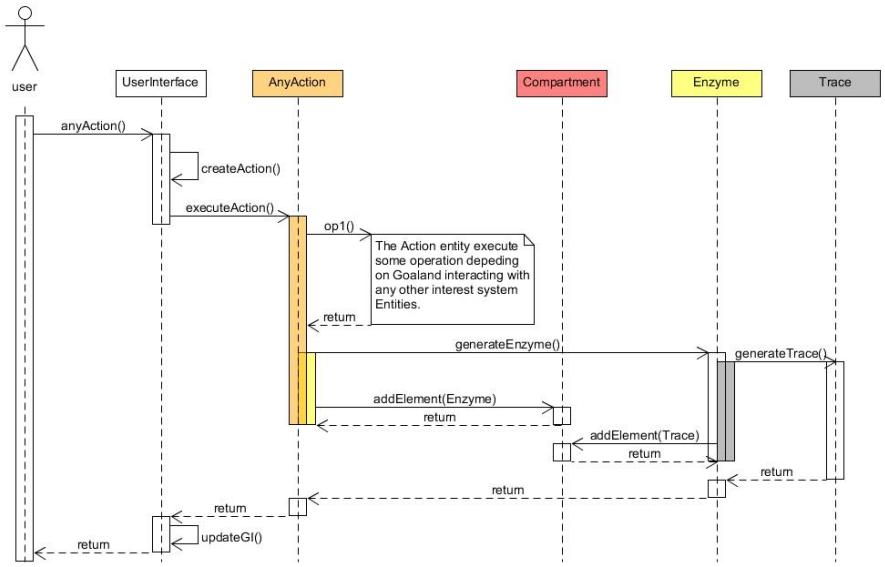

Figura 4.2: Schema di interazione fra le entità del sistema a seguito dell'esecuzione di una qualsiasi *Action*

Capitolo 5

Perturbation

5.1 Perturbation

In questa ultima sezione, verrà ulteriormente esteso il modello del dominio per integrare al suo interno anche l'entità di Perturbation. Le *Perturbation* sono delle azioni implicite eseguite dal sistema MoK in modo trasparente all'utente, tuttavia la perturbazione effettuata dipende dal particolare stato del sistema. Quest'ultimo può essere alterato attivamente dall'utente o dalle perturbazioni stesse.

5.1.1 L'insieme di Perturbation

Similmente alle *Action*, le *Perturbation* sono un insieme di azioni eseguite dal sistema che possono modificare lo stato delle altre entità che ne fanno parte.
Il lavoro del Dottor Mariani ha permesso di individuare un insieme di possibili *Perturbation* valide per un qualsiasi sistema MoK, le quali verranno di seguito elencate.

- **Approach**: quest'azione permette di avvicinare due *Compartment* e può essere causata da una rilevazione dell'aumento di scambio di informazioni fra gli stessi;
- **Attract**: permette di avvicinare o spostare in uno specifico *Compartment* tutte le entità che rispettano un particolare grado di similari-

tà con un data entità Target. La perturbazione contraria invece è chiamata **Repulse**;

- **Boost**: aumenta il ~~rateo~~ di generazione di Reaction MoK per maggiori dettaglio si faccia riferimento a [1]. La perturbazione contraria, invece, è definita come **Wane**;

- **Strengthen**: questa perturbazione aumenta la forza delle correlazioni di tutte le entità simili a un particolare Target.

L'elenco delle Perturbation può ovviamente essere ampliato per soddisfare tutte le esigenze di un particolare sistema MoK, ad esempio è possibile considerare l'introduzione della pertubation **Weaken** opposta alla perturbazione Strengthen.

5.1.2 Il dominio esteso

Come rappresentare e gestire queste *Perturbation*?
Il modello delle Perturbation rispecchia la rappresentazione adottata per le *Action*, infatti è stata implementata una classe astratta: *Perturbation*.
Questa classe rappresenta gli attributi e le funzioni comuni a tutte le Perturbation:

5.1.2.1 Perturbation - Attributi

- **species**: è l'attributo che determina il tipo di Perturbation (Attract,Repulse,Approach,...);

- **target**: rappresenta la famiglia di entità che viene influenzata dalla *Perturbation*, le altre entità non vengono modificate;

- **executed**: è un flag che indica se la Perturbation è già stata eseguita oppure no, l'utilizzo di questo flag verrà descritto in dettaglio più avanti;

- **position**: è il compartimento in cui esiste la Perturbation;

- **perturParam**: similmente alla Map context utilizzata per altre entità, perturParam può contenere tutte le variabili che possono essere

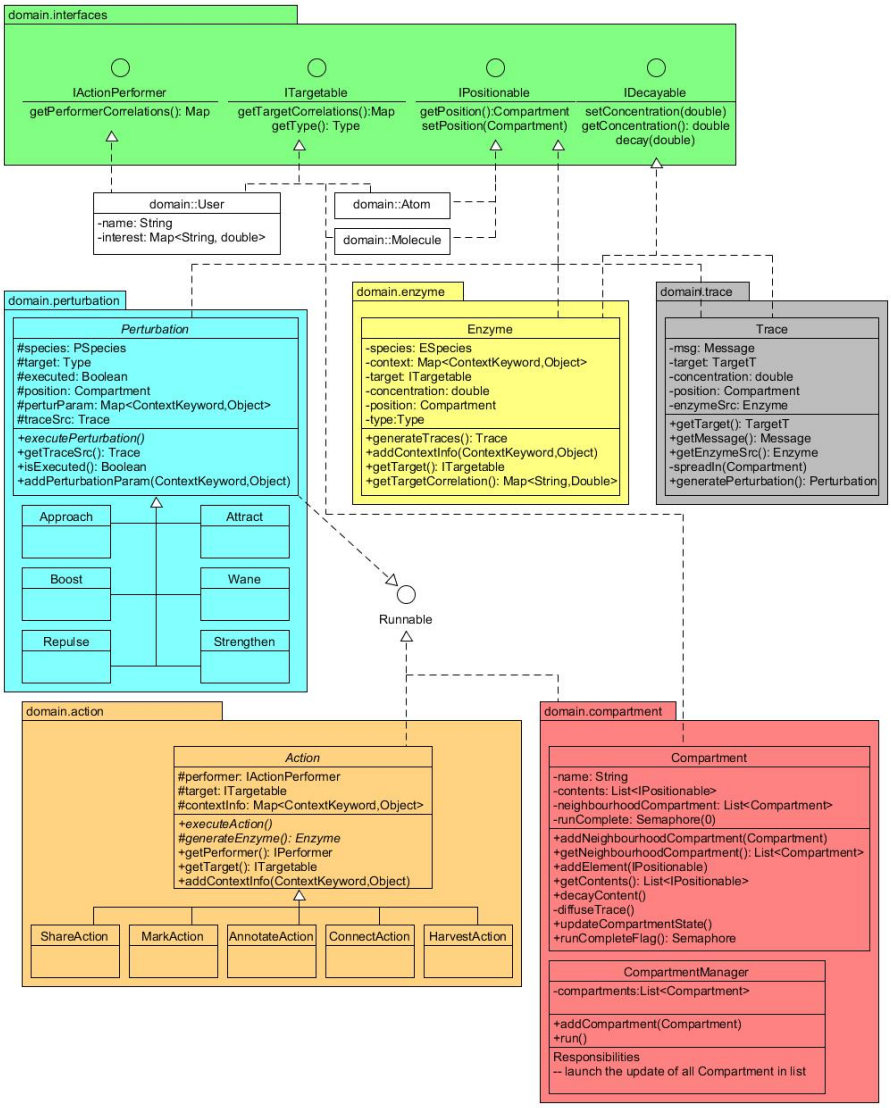

Figura 5.1: Modello del dominio esteso con l'aggiunta delle Perturbation e delle entità necessarie alla loro gestione

utili all'esecuzione della Perturbation, ad esempio: range di similarità, entità per il confronto di similarità oppure ~~ratei~~ di rinforzo o indebolimento;

- **traceSrc**: poichè le Perturbation sono generate dalle **Trace** presenti nei *Compartment* del sistema, si tiene un riferimento all'entità generatrice.

5.1.2.2 Perturbation - Funzioni

- **executePerturbation**: è la funzione Astratta che poi dovrà essere implementata in modo opportuno dalle varie azioni di *Perturbation*

- **isExecuted**: modifica il flag executed impostando a True, notificando così l'esecuzione avvenuta della *Perturbation*.

5.1.2.3 Trace

L'entità *Trace* è colei che può generare una Pertubation, la quale dipende dalla coppia *Enzyme-Trace* in particolare dalla species dell'*Enzyme* e dal msg della *Trace*. Per questo motivo all'entità *Trace* è stata aggiunta la funzione generatePerturbation(), la quale in base all'enzima e al contesto analizzato creerà una perturbazione opportuna.

5.1.3 La similarità

Molte azioni di perturbazione hanno effetto solo su Target considerati simili al bersaglio dell'azione che ha generato la *Perturbation*. Esistono diversi indici per quantificare la similarità fra due elementi, nel progetto è stato utilizzato il seguente:

$$Sim(C1, C2) = \frac{\sum_{k=1}^{n} c_i(t_k) * c_j(t_k)}{\sqrt{\sum_{k=1}^{n} c_i(t_k)^2 * \sum_{k=1}^{n} c_j(t_k)^2}}$$

Dove C_i e C_j sono i vettori dei pesi dei documenti da comparare e t_k rappresenta la k-esima parola del documento.
La funzione di similarità è implementata nella libreria *SimilarityLib* e utilizzata dalle entità opportune. Si noti che la scelta della funzione è arbitraria

ed implementata solo per effettuare alcuni test sul funzionamento dell'architettura. Non essendo parte fondamentale del sistema, può essere sostituita da qualsiasi altre funzione di similarità.

5.1.4 Lo stato del sistema

In precedenza, è stato ripetutamente citato il concetto di Stato del sistema, fondamentale per entità che necessitano di analizzare il contesto o lo stato del sistema per eseguire correttamente le proprie mansioni, come: *Enzyme*, *Trace* e *Perturbation*. Queste, infatti, necessitano di analizzare il contesto o stato del sistema per eseguire correttamente le proprie mansioni. Nell'architettura presentata lo stato del sistema è rappresentato dall'insieme degli stati di ogni *Compartment*. A sua volta, lo stato del *Compartment* è definito dalla quantitò e dal tipo di *Enzyme*, *Trace* e *Perturbation* contenute al suo interno.
L'architettura considera quindi i Compartment come entità attive che a cicli regolari analizzano e modificano di conseguenza il loro stato, operazione che comprende l'esecuzione della decay sulle entità *IDecayable*.

5.1.4.1 Il CompartmentManger

La coordinazione dei cicli di aggiornamento è demandata a una nuova entità: il *CompartmentManager* il quale si occupa di scandire il ciclo di aggiornamento per un gruppo di *Compartment*. Si noti come il raggruppamento di *Comparment* all'interno di un *ComparmentManager* è puramente logico e come i gruppi possono essere formati in base a come meglio si adattano alle necessità del sistema.
Il *CompartmentManager*, inoltre, implementa un Executor al suo interno e considera tutte i **Compartment** che gestisce come Task Runnable da eseguire. In un'ottica distribuita, si può quindi vedere il *CompartmentManager* come nodo di un Cluster con le sue risorse elaborazionali e relativi *Compartment* da gestire, i quali infatti sono visti come Task da eseguire.

5.1.5 Modello di Interazione

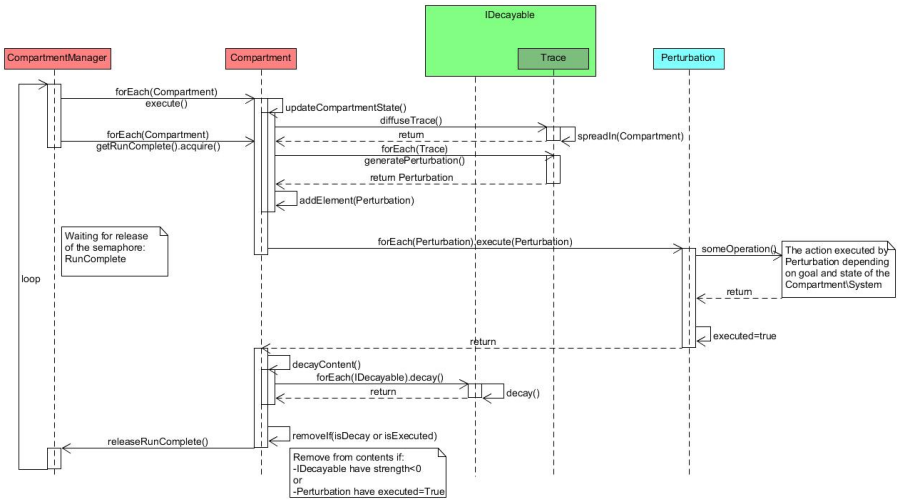

Figura 5.2: Schema di interazione durante il ciclo di aggiornamento di un Compartment

5.1.6 Problematiche di Concorrenza

L'architettura proposta mette in evidenza alcune problematiche legate all'esecuzione concorrente dell'aggiornamento dei Compartment gestiti da un particolare *CompartmentManager*.
Il primo problema affrontato è la sincronizzazione delle procedure di aggiornamento: prima di concludere e ricominciare un ciclo di aggiornamento, il *CompartmentManager* deve infatti attendere che tutti i *Compartment* da lui gestiti abbiano completato la loro procedura.
Questa problematica è stata risolta con l'aggiunta di un semaforo Evento per ogni *Compartment*, il quale viene rilasciato solo al termine della procedura di aggiornamento del *Compartment*.
L'accesso a risorse condivise è un altro grosso problema di concorrenza. Le azioni eseguibili sulle entità del sistema possono entrare in conflitto per le stesse risorse, quindi è opportuno gestire l'accesso di conseguenza.
L'accesso alle risorse da parte delle azioni di *Perturbation* è stato ridotto il più possibile, consentendo solo al *Compartment*di modificare i propri contenuti. Le poche eccezioni riguardano:

- accesso in lettura a Entità di riferimento come Enzimi o Tracce generanti;
- ~~aggiunta di elementi a un *Compartment* causato ad esempio dalla *Attract - Perturbation*;~~
- ~~accesso a librerie di utilità per il sistema.~~

Bibliografia

[1] S. Mariani *Coordination Issues in Complex Socio-Technical Systems: Self-Organisation of Knowledge in MoK*, 2016.