
Università degli Studi di Firenze

Scuola di Ingegneria - Dipartimento di Ingegneria dell'Informazione


---

Tesi di Laurea Triennale in Ingegneria Informatica

# Analisi e sviluppo di un componente Java per la generazione automatica di modelli NetLogo


*Candidato*
Aurel Pjetri

*Relatore*
Prof. Enrico Vicario

*Co-relatore*
Dott. Sandro Mehic


---


Anno Accademico 2015/2016

# Indice

# Introduzione

La simulazione delle masse negli ultimi decenni ha attirato l'attenzione di un numero crescente di gruppi di ricerca. Gli ambiti in cui questo fenomeno trova applicazioni interessanti sono molteplici: scienza della sicurezza (evacuazione di ambienti pubblici), progettazione architetturale, studio del flusso di traffico, studio sociologico dei comportamenti collettivi e anche intrattenimento.

La dinamica delle folle è di grande interesse sia sotto condizioni critiche che normali. Nella fase di progettazione di ambienti chiusi come centri commerciali, stadi e scuole dove centinaia di persone si concentrano per scopi differenti bisogna garantire una via di uscita a tutti gli utenti nonostante il numero limitato di punti di uscita. In condizioni normali le simulazioni possono essere utilizzate per studiare il comportamento delle folle in ambienti come fiere, quartieri, o intere città, in modo, ad esempio, da agevolare la viabilità nei tratti più affollati.

Lo studio delle masse è un argomento molto affascinante, ma allo stesso tempo, molto complesso. Per simulare situazioni del mondo reale è spesso richiesta la modellazione di comportamenti collettivi, come anche individuali, di folle di grandi dimensioni che si muovono in ambienti anch'essi molto grandi. La complessità quindi si sviluppa in due direzioni: logica e temporale. Le simulazioni di modelli di grandi dimensioni, infatti, sono processi che

richiedono molto tempo e grande potenza computazionale. Come è possibile ridurre il tempo di esecuzione di queste simulazioni?
Un possibile approccio, per risolvere questa criticità, è integrare la simulazione vera e propria con metodi analitici, che possono usare vari formalismi per modellare il sistema, fra i quali le Catene di Markov. Si scompone il modello in porzioni di dimensioni minori e si eseguono simulazioni isolate su ognuno di questi, quindi si usano metodi matematici e analitici per mettere insieme questi risultati ed ottenere dei dati che cercano di avvicinarsi il più possibile a quelli che si sarebbero ottenuti eseguendo la simulazione sul modello completo.
Il vantaggio di questo approccio modulare è la possibilità di cambiare alcune parti della mappa e ottenere nuovi risultati eseguendo simulazioni isolate sulle patch coinvolte dai cambiamenti invece di dover eseguire nuovamente il modello monolitico.
L'obiettivo perseguito in questa tesi quindi è analizzare e sviluppare un framework per la generazione automatica e ed esecuzione delle simulazioni dei modelli NetLogo, per ridurre i tempi di simulazione in un contesto dinamico e facilitare la modellazione dei sistemi di movimento delle masse.
In concreto il progetto consiste nello sviluppo di un componente Java che riceve in ingresso un documento XML in cui è descritto il modello suddiviso in topologia, comportamenti possibili degli attori e stato iniziale dell'ambiente. In seguito alla costruzione degli oggetti Java necessari per la rappresentazione del modello viene eseguita la scrittura del codice NetLogo che modella l'ambiente inizialmente descritto. In questo modo si evita la scrittura da parte di un umano della parte del codice NetLogo che risulta essere meccanica e ripetitiva, ed invece si usa un documento XML per raccogliere le informazioni necessarie alla sua generazione, che allo stesso tempo rende fruibili i dati da

altri ambienti di sviluppo .

Il presente progetto è stato sviluppato presso il Software Science and Technology Laboratory (STLAB), Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Firenze, sotto la supervisione del professor Enrico Vicario e del dottor Sandro Mehic.

# Capitolo 1

# Simulazione delle masse

In questo capitolo si descrive il contesto di ricerca in cui si inserisce questo progetto di tesi, andando a descrivere, nella Sezione 1.1, lo stato dell'arte della simulazione delle masse e, nella Sezione 1.2, l'approccio di ricerca in cui questo progetto di tesi si è reso necessario.

Come già accennato nell'introduzione, la simulazione delle masse è un ambito di ricerca che negli ultimi anni ha guadagnato molta rilevanza. Questo grazie a un miglioramento delle capacità computazionali e dei metodi utilizzati, ma anche a causa di eventi sociali sempre più frequenti e di dimensioni sempre maggiori. Quindi diventa sempre più importante la capacità di prevedere i movimenti delle masse in modo da garantire la sicurezza delle persone in ogni tipo di situazione.

Le criticità che rendono la modellazione delle masse un problema complesso sono le grandi dimensioni dei modelli e i comportamenti individuali, sia sociali che fisici, che devono essere presi in considerazione.

Descriviamo brevemente gli approcci più diffusi e riconosciuti per la simulazione delle masse per andare poi a soffermarci sull'approccio gerarchico in cui si inserisce questo progetto di tesi.

## 1.1 Stato dell'arte

Nella modellazione delle masse il maggior problema è la dimensione del modello da studiare. Le risposte ideate per risolvere questa criticità sono state molte e possono essere divise in due categorie sulla base del loro approccio: su scala microscopica e su scala macroscopica.

### 1.1.1 Approccio su scala microscopica

L'approccio su su scala microscopica si basa sull'idea di rappresentare ogni persona come agente o macchina a stati finiti. I quattro approcci più usati sono: *social force model*, *cellular automata*, *magnetic force model* e *rule-based model*.

Il social force model [8] si fonda sul concetto che le forze sociali possono essere rappresentate come forze meccaniche che agiscono tra le persone. Le forze prese in considerazione sono repulsione, attrito e dissipazione. Solitamente questi modelli arrivano ad essere molto espressivi permettendo di descrivere comportamenti sia fisici che sociali nel modello.

Il magnetic force model [12] è simile al precedente approccio, questo infatti introduce il concetto di forze magnetiche per modellare i comportamenti degli individui. Persone e ostacoli vengono modellati come poli positivi, mentre i punti di interesse, come ad esempio le uscite, sono rappresentati come poli negativi. In questo modo gli agenti sono attratti dai punti di interesse e respingono ostacoli e altre persone evitando le collisioni.

L'approccio cellular automata [7] è caratterizzato da una griglia bidimensionale dove una cellula è definita dal proprio stato, che evolve nel tempo sulla base del proprio valore e dello stato delle cellule adiacenti. Modificando

il numero di cellule che influenzano lo stato si possono rappresentare comportamenti diversi, ad esempio per il contatto fisico si usa un insieme molto ristretto di cellule, al contrario per il campo visivo si usa un numero maggiore di cellule.
Infine il rule-based model [14] presenta una modellazione dello spazio e del tempo continui e prevede che velocità e direzione di un attore vengano modificate sulla base di determinate regole, come conformismo, anticonformismo e coesione.

### 1.1.2 Approccio su scala macroscopica

L'approccio su macro-scala basa il proprio studio sull'analisi di caratteristiche macroscopiche della massa come la densità media e la velocità. Questa analisi viene condotta attraverso lo studio di equazioni differenziali che descrivono l'evoluzione della massa, oppure attraverso algoritmi di apprendimento e regressione applicati su dati esistenti.
Gli approcci più diffusi su scala macroscopica sono: *regression model*, *queuing model*, *fluid dynamics*.
I regression models si basano su dati empirici dai quali cercano di estrapolare una funzione in grado di descrivere l'evoluzione della folla.
L'approccio fluid dynamics, invece, rappresenta gli agenti come particelle di un fluido e studia quindi le equazioni differenziali che ne descrivono i cambiamenti nel tempo.
Oltre al problema della definizione di modelli scalabili, esiste un ulteriore punto critico rappresentato dalla mancanza di dati empirici e dalla difficoltà di condurre esperimenti su scale macroscopiche.
Molti modelli per eseguire queste simulazioni aggregano singoli agenti in

gruppi dallo stesso comportamento, introducendo quindi comportamenti collettivi e riducendo il numero di agenti da modellare. Questa soluzione però non elimina la dipendenza del costo computazionale dal numero di attori, quindi, nonostante questi modelli siano in grado di lavorare con un numero elevato di attori, non sono ottimali per lo studio di intere città, che coinvolge milioni di agenti.

## 1.2 Approccio gerarchico

L'approccio proposto da E. Vicario, S. Mehic e M. Paolieri in [11] per affrontare il problema della simulazione delle masse, rispetto alle tipologie precedentemente esposte, è di tipo ibrido. Si scompone la simulazione in tre livelli di scala diversi in modo da ottenere una soluzione analitica indipendente dal numero di agenti.

Al primo livello si usa una modellazione di tipo microscopico basata su agenti (Agent Based Modeling, ABM). In questa fase della ricerca si rende piuttosto utile il presente progetto, che facilita la generazione dei modelli NetLogo per l'esecuzione delle simulazioni e l'estrazione di parametri come tempo medio di soggiorno e probabilità di transizione.

L'agent-based modeling, oltre alle strategie di movimento fisico, permette l'inclusione di una serie di altri aspetti, in particolare quello sociale della folla.

Un comportamento di particolare interesse è l'altruismo, ovvero la probabilità che un attore aiuti un altro attore con una capacità di movimento minore della sua. É stato dimostrato infatti che questo risulta essere un aspetto caratteristico delle masse.

Un altro behavior che risulta interessante è il conformismo, ovvero quanto

questo attore tenderà ad unirsi alla massa oppure a cambiare percorso di fronte a una strada affollata.
La strategia di movimento rappresenta l'aspetto fisico dell'attore e consiste nell'individuazione e raggiungimento dei punti di interesse e dell'uscita più vicina.
Attraverso la combinazione di questi due aspetti si riescono ad ottenere comportamenti molto complessi che rispecchiano l'andamento della folla nella realtà.
Il secondo livello di questo approccio, ovvero il livello mesoscopico, consiste nell'astrazione dei singoli agenti delle precedenti simulazioni (*Agent Based Simulations*, ABS) in catene di Markov tempo-discrete modellate usando i dati raccolti dalle ABS.
Ogni catena di Markov tempo-discreta, quindi, rappresenta un singolo agente in una particolare regione dello spazio e i comportamenti precedentemente osservati vengono usati per definire le probabilità di transizione nella catena individuale.
Nel terzo ed ultimo livello, quello macroscopico, si compongono le N catene di Markov parallele e le M regioni spaziali adiacenti. Quindi il macroscopico spazio fisico ottenuto non è altro che un grafo con le varie regioni come nodi e i loro collegamenti come archi.
Come spiegato nel report di ricerca [11], il passo più importante di questa fase è l'approssimazione di campo medio. Usando infatti la *fast simulation technique* presentata in [10] si è in grado di rimuovere la dipendenza del costo computazionale delle simulazioni dal numero di agenti coinvolti, permettendo, quindi, di ottenere una soluzione analitica per la simulazione di una massa che è indipendente dalle sue dimensioni.
In conclusione il problema del costo computazionale è risolto con l'introdu-

zione dell'approssimazione di campo medio e la dipendenza dal numero di agenti è risolta con l'uso delle catene di Markov che li sostituisce. Inoltre il numero degli stati di queste catene è mantenuto basso grazie all'approccio gerarchico impostato.

# Capitolo 2

# NetLogo e DesignPattern

In questo capitolo si fornisce una descrizione dell'ambiente NetLogo e dei Design Patterns che questo progetto usa al fine di una migliore comprensione e chiarezza dei concetti esposti nel Capitolo 3.

## 2.1 NetLogo

NetLogo [15] è un linguaggio di programmazione e un ambiente di modellazione di sistemi complessi. Adatto per la simulazione e lo studio di fenomeni naturali e sociali che si evolvono nel tempo.
Gli utenti possono dare istruzioni a migliaia di agenti che operano in modo concorrenziale. Questi comandi possono essere dati in modo individuale o collettivo, permettendo quindi lo studio dei comportamenti su più livelli, come quello microscopico dei behaviors individuali o quello macroscopico delle loro interazioni con gli altri.
NetLogo è usato per costruire una infinita varietà di simulazioni. I suoi turtles sono stati trasformati in molecole, lupi, clienti, commercianti, api, uccelli, batteri, macchine, magneti, pianeti, formiche e molto altro. Le sue patches

allo stesso modo sono state usate come alberi, muri, corsi d'acqua, cellule tumorali, cellule vegetali e altro ancora. Turtles e patches possono essere usate per studiare astrazioni matematiche, ma anche per fare arte o giocare. Tra i temi studiati con questo strumento ci sono automi cellulari, algoritmi genetici, evoluzione, ottimizzazione e individuazione di percorsi, dinamiche della popolazione e società artificiali.
Tutti questi modelli condividono il topic centrale perseguito da questo strumento che sono i sistemi complessi e i comportamenti emergenti.
Il più grande miglioramento apportato a partire dalla versione 2.0 in poi riguarda la grafica. In particolare i turtles possono essere di qualsiasi forma e dimensione e soprattutto possono essere posizionati in qualsiasi punto dello spazio. La loro grafica è di tipo vettoriale in modo da non avere perdita di qualità dell'immagine a qualsiasi scala si visualizzi il modello.
Uno dei principali obbiettivi di NetLogo è quello di essere scientificamente riproducibile, quindi i suoi modelli operano in modo deterministico. Inoltre le librerie matematiche Java platform-indipendent su cui si basa aiutano a dare consistenza e a rendere NetLogo indipendente dalla macchina su cui viene eseguito.

### 2.1.1 Storia

Nasce a scopo educativo e di ricerca, dalla fusione di **Logo** [13] e **StarLisp** [9]. Dal primo eredita il principio *"low threshold , no ceiling"*, ovvero bassa soglia di conoscenza per il suo utilizzo, rendendolo accessibile a utenti inesperti nella programmazione, ma allo stesso tempo completa programmabilità, rendendolo quindi anche uno strumento utile per la ricerca.
Da Logo viene ereditato anche il concetto fondamentale di *turtle*, con la diffe-

renza che Logo permetteva il controllo di un unico agente, mentre un modello NetLogo può averne migliaia. Da StarLisp, invece, NetLogo eredita i molteplici agenti e la loro *concurrency*.
Il design di NetLogo si basa sulla precedente esperienza dei suoi creatori con StarLogoT [16], di cui sono stati rielaborati quasi interamente sia l'interfaccia che il linguaggio, aggiungendo soprattutto funzionalità destinate a utenti nel campo della ricerca.

### 2.1.2 Linguaggio

Come linguaggio NetLogo si evolve da Logo al quale aggiunge il concetto di agenti e di concurrency. In generale Logo è molto conoscuto per il concetto di `turtle` che ha introdotto. NetLogo generalizza questo permettendo il controllo di centinaia o migliaia di turtles che si muovono e interagiscono tra di loro.
Il mondo in cui i turtles si muovono è suddiviso in `patches` anche esse interamente programmabili e identificabili attraverso coordinate spaziali.
Turtles e patches vengono chiamate collettivamente `agents`. Tutti gli agenti possono interagire tra di loro e eseguire istruzioni in modo concorrenziale.
NetLogo include inoltre un terzo tipo di agente, l'`observer` il quale è unico. In generale l'observer è quello che impartisce ordini agli agenti.
Possono essere definite diverse "razze" (`breeds`) di turtles, ciascuna con variabili proprie. In questo modo è possibile definire comportamenti caratteristici e distinti per ogni breed definita.
Una peculiarità che contraddistingue NetLogo dai suoi predecessori sono gli "*agentsets*", ovvero insiemi di agenti componibili anche al volo nel momento del bisogno. Questa caratteristica ha aumentato notevolmente la capacità espressiva del linguaggio in quanto permette di selezionare insiemi di agenti,

anche eterogenei, che soddisfano condizioni arbitrarie a cui impartire istruzioni specifiche.
Si possono dichiarare tre tipi diversi di variabili: globali, locali e proprietarie. Le variabili globali possono essere modificate da qualsiasi parte del codice e vengono dichiarate fuori da qualsiasi procedura. Le variabili locali, invece, hanno visibilità limitata alla procedura in cui vengono dichiarate. Le variabili proprietarie, infine, sono associate esclusivamente alle turtles o patches oppure a una specifica breed di esse.
Le procedure sono definite tutte in un unico file insieme alle variabili e sono destinate all'esecuzione da parte di specifici agenti oppure dell'observer stesso. Si possono definire procedure reporter o subroutine, dette anche command. Entrambe possono ricevere parametri in ingresso, la loro differenza sta nel fatto che i reporters, al contrario delle subroutine, restituiscono un valore in uscita. Quindi mentre le prime vengono usate per calcolare determinati valori e reperire specifiche informazioni, le seconde vengono usate per impartire istruzioni ad agenti e produrre effetti su variabili e altri agenti.
Concludendo, il linguaggio presenta tutti i costrutti standard come procedure, loop, liste, integrati con particolari costrutti per il supporto alla modellazione multi-agente.

## 2.2 Pattern Builder

Lo scopo di questo pattern è quello di separare la costruzione di un oggetto complesso dalla sua rappresentazione, in modo che il medesimo processo possa essere usato per creare rappresentazioni diverse.

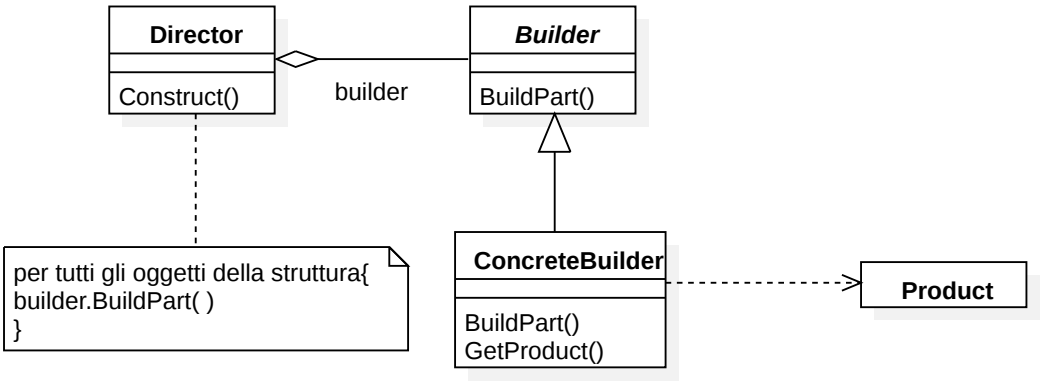


Figura 2.1: Class diagram di un generico Builder

### 2.2.1 Applicabilità

Il pattern Builder è solitamente usato nei seguenti casi:

- l'algoritmo per la creazione di un oggetto deve rimanere separato dalle parti che lo costituiscono e dal modo in cui esse sono composte;
- il processo di costruzione deve rendere possibili diverse rappresentazioni dell'oggetto a cui è associato.

### 2.2.2 Partecipanti

Il Builder coinvolge le seguenti classi:

- **Builder** mette a disposizione un interfaccia per la costruzione delle parti dell'oggetto '*Product*'
- **ConcreteBuilder**
  - Costruisce le parti dell'oggetto e le assembla secondo una specifica rappresentazione.
  - Definisce e tiene traccia delle varie rappresentazioni create.
  - Espone un metodo per ottenere il Product creato.

- **Director** sfrutta l'interfaccia del Builder per costruire il Product.
- **Product** è l'oggetto complesso che viene costruito. ConcreteBuilder provvede a costruire le sue parti e definisce il processo con cui vengono assemblate.

### 2.2.3 Collaborazioni

Il tipico funzionamento del Builder comporta le seguenti cooperazioni:

- Il Client mette in vita il Director e lo configura con il Builder corrispondente alla rappresentazione che desidera costruire.
- Director manda una richiesta ogni volta che una parte del Product deve essere costruita.
- Builder riceve le richieste del Director, costruisce e compone le parti del Product.
- Il Client recupera dal Builder il Product costruito.

### 2.2.4 Conseguenze

L'introduzione del Builder in un progetto software produce le seguenti conseguenze:

1. Consente di variare la rappresentazione interna di un prodotto. Il Builder infatti, grazie alla sua interfaccia, nasconde la struttura interna dell'oggetto che costruisce, quindi per modificarla basta usare una diversa implementazione di questa interfaccia.

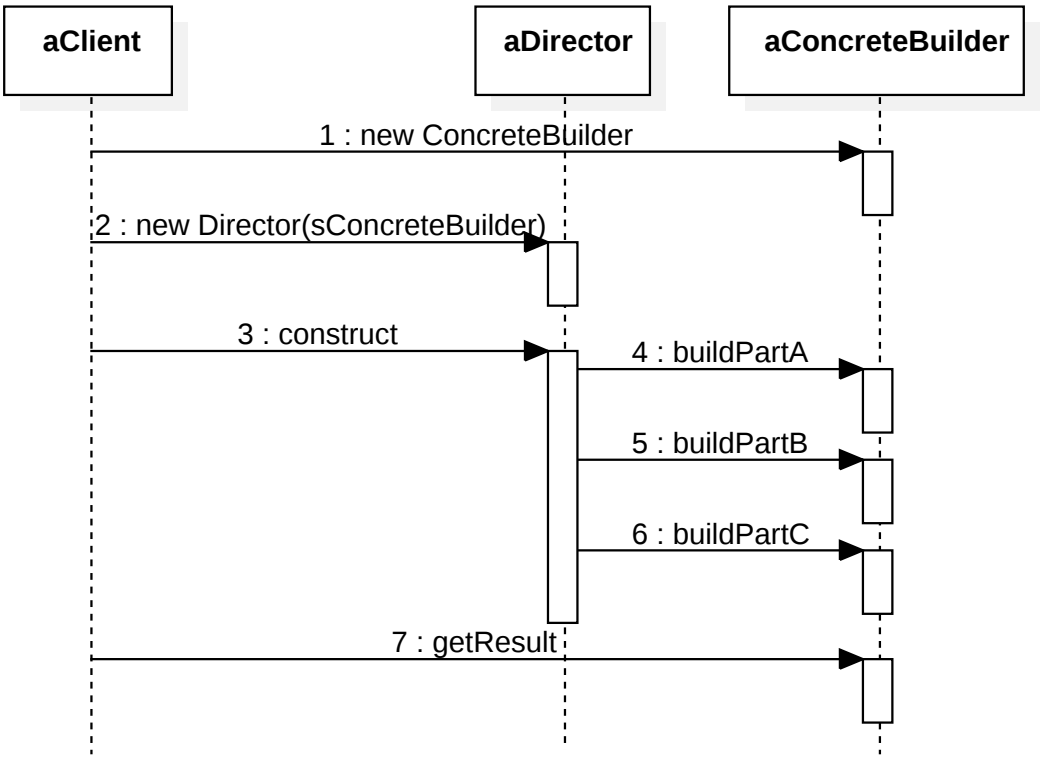


Figura 2.2: Sequence Diagram di un generico Builder

2. Migliora la modularizzazione incapsulando il codice necessario alla costruzione dei prodotti. In questo modo i client non hanno bisogno di conoscere la struttura interna dei prodotti. Inoltre Director diversi possono costruire varianti del prodotto con le stesse tipologie di parti interne.

3. Fornisce un controllo sul processo di costruzione molto ravvicinato. Al contrario di altri pattern creazionali che restituiscono il prodotto in un 'colpo' solo, questo segue il processo passo per passo rendendolo molto più controllato e sicuro.

## 2.3 Pattern Visitor

Visitor è uno pattern molto utile che consente di aggiungere nuove funzionalità alle parti di un oggetto composito senza modificare gli oggetti stessi.

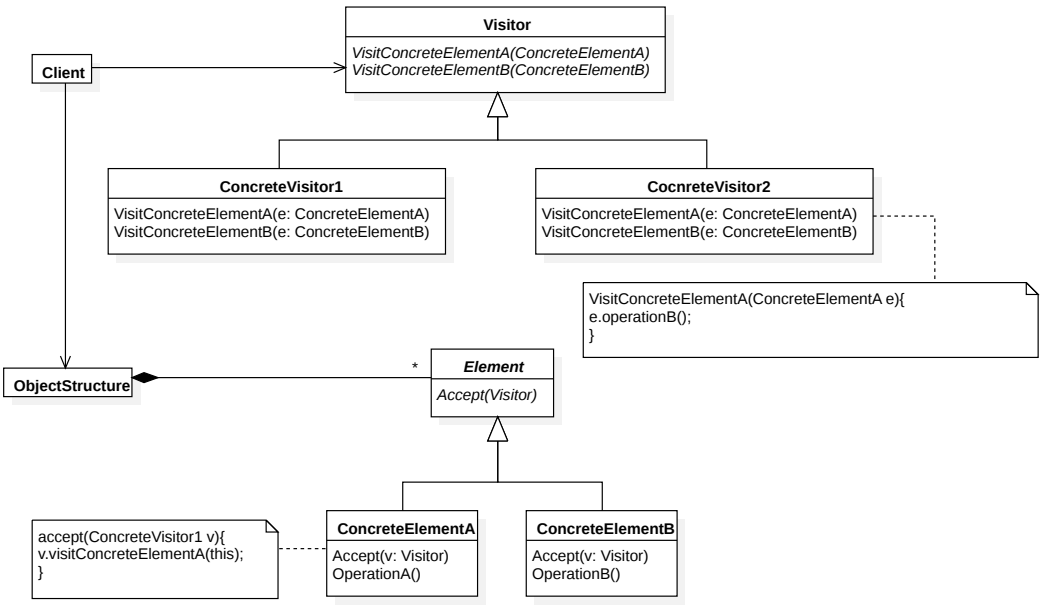


Figura 2.3: Class Diagram di un generico Visitor

### 2.3.1 Applicabilità

L'utilizzo del pattern Visitor è consigliato nelle seguenti situazioni:

- Quando si devono eseguire operazioni diversificati in base alla classe concreta degli oggetti che compongono una struttura, ma questi presentano interfacce diverse.
- Quando devono essere aggiunte operazioni su alcuni oggetti evitando di inquinare le loro interfacce.
- Quando si ha un frequente bisogno di aggiungere funzionalità a classi che invece cambiano di rado. Infatti nel caso in cui queste classi dovessero cambiare, si dovrebbero modificare e aggiornare tutti i Visitor ad esse associati.

### 2.3.2 Partecipanti

Nell'implementazione di questo pattern vengono coinvolte le seguenti classi:

- **Visitor** definisce un metodo per ogni ConcreteElement, in modo tale da essere in grado di capire il tipo concreto dell'oggetto che ha chiamato il metodo.
- **ConcreteVisitor** implementa tutti i metodi definiti da Visitor. Fornisce inoltre un contesto per l'algoritmo a cui è associato tenendo memoria di eventuali risultati parziali.
- **Element** definisce il metodo `Accept()` con cui viene chiamata l'operazione del Visitor.
- **ConcreteElement** implementa il metodo `Accept()`.

- **ObjectStructure** può fornire un'interfaccia utile al Visitor per attraversare la struttura.

### 2.3.3 Collaborazioni

Di seguito descriviamo le interazioni che avvengono tra le classi coinvolte per la realizzazione del pattern, come si vede anche in Figura 2.4.

- Il Client crea un oggetto di tipo ConcreteVisitor e attraversa la struttura di oggetti con questo.
- L'oggetto che viene visitato dal Visitor chiama il metodo per la visita corrispondente alla propria classe concreta passando a se stesso come parametro.

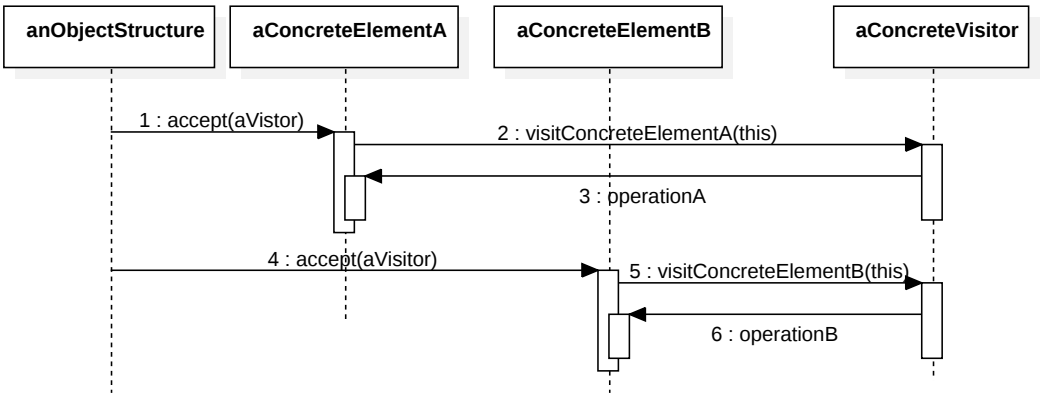


Figura 2.4: Sequence Diagram di un generico Visitor

### 2.3.4 Conseguenze

Gli effetti che il Visitor ha su un progetto software sono molteplici:

1. Visitor consente di aggiungere facilmente operazioni a oggetti complessi. Infatti per definire una nuova operazione basterà definire un nuovo

Visitor, senza quindi andare a modificare tutte le classi degli oggetti che compongono la struttura nel caso in cui questa operazione coinvolga più di uno di questi oggetti.

2. Aiuta a mantenere le operazioni correlate sotto la stessa classe e a separare quelle scorrelate.

3. Complica l'aggiunta di classi ConcreteElement, poiché questo implica l'aggiornamento di tutte le interfacce dei Visitor che devono poter visitare l'elemento. Quindi è auspicabile usare questo pattern nei casi in cui la struttura rimane invariata o comunque cambia con frequenza minore rispetto alle operazioni da eseguirvi.
   Nei casi in cui la struttura di classi cambia frequentemente può diventare conveniente aggiungere le operazioni direttamente dentro queste classi.

4. Visitor al contrario di altri pattern come l'*Iterator* non è vincolato a visitare oggetti appartenenti alla stessa classe.

5. Il funzionamento del Visitor è tale da imporre all'elemento concreto di implementare metodi pubblici che accedono al loro stato interno compromettendo quindi il loro incapsulamento.

6. Il Visitor ha il vantaggio di tenere traccia dell'attraversamento della struttura cosa che altrimenti deve essere passata tra i parametri dei vari metodi eseguiti sugli oggetti.

# Capitolo 3

# Sviluppo del progetto

L'obiettivo di questa tesi è realizzare un programma in linguaggio Java che sia in grado di ricevere in ingresso la struttura del modello descritta in linguaggio XML e di scrivere in modo automatico il codice NetLogo che possa eseguire la simulazione di interesse, rendendo, quindi, trasparente il processo di scrittura del codice.

Questo progetto è costituito da una struttura composita centrale di classi in cui vengono conservate tutte le informazioni necessarie per l'esecuzione della simulazione. Come mostrato nella Sezione 3.1 l'oggetto di tipo Graph che rappresenta la struttura è composto da classi che modellano la topologia dell'ambiente e da classi che rappresentano i comportamenti da simulare.

L'informazione conservata in questa struttura viene estratta da un apposito parser che implementa il pattern Builder per la costruzione e composizione degli oggetti.

Infine si è implementato il pattern Visitor per visitare gli oggetti della struttura e generare il relativo codice NetLogo.

I modelli NetLogo che vengono generati da questo componente sono costituiti da un ambiente fisico modellato sotto forma di grafo con nodi e archi

che rappresentano rispettivamente incroci e strade, caratterizzati da coordinate spaziali e dimensioni fisiche. In questo ambiente si muovono attori che possono avere comportamenti differenti. Nel nostro caso abbiamo deciso di modellare i comportamenti degli attori come una lista di punti di interesse che questi devono visitare prima di poter uscire.

Nella Sezione 3.4 andiamo a delineare la struttura dei documenti XML previsti e nella Sezione 3.5 si descrive brevemente il funzionamento del relativo parse. Nella Sezione 3.6 invece tracciamo la struttura del modelli NetLogo generati. Infine nella Sezione 3.7 si descrivono le librerie usate.

## 3.1 Struttura centrale

L'unico scopo delle classi Java utilizzate è quello di rappresentare e conservare l'informazione raccolta dal documento XML, senza eseguire alcun tipo di manipolazione. Per questo motivo abbiamo cercato di mantenere la struttura delle classi il più semplice possibile, come mostrato in Figura 3.1.

La classe Graph è caratterizzata da una composizione di Edge, Node e Behavior. Espone metodi per il recupero delle informazioni e per la validazione di nodi e archi, in modo tale da impedire l'aggiunta di nodi nella medesima posizione e di archi con gli estremi uguali.

Ogni behavior è caratterizzato da un identificatore, attribuitogli dall'XML e da una lista di nodi regolari di interesse che devono essere visitati prima di permettere all'attore di puntare ad una uscita.

Gli edges in questo modello sono aggregazioni di due nodi e presentano un particolare attributo "weight" che verrà usato dalle funzioni NetLogo per la generazione dei percorsi a costo minimo.

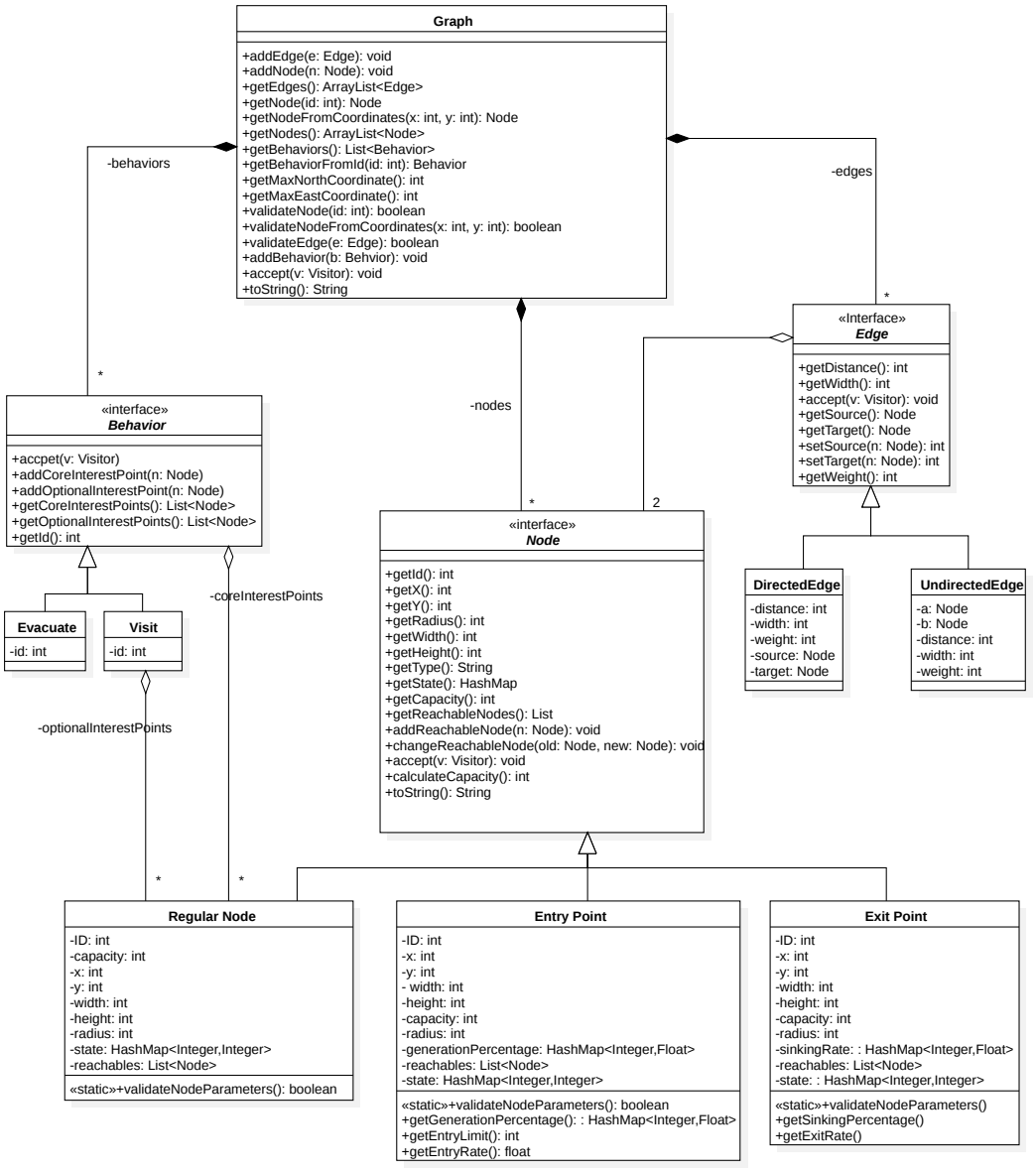


Figura 3.1: Class diagram della struttura del grafo

Infine ogni classe che implementa l'interfaccia Node, oltre ai metodi per il recupero delle informazioni conservate, implementa anche un metodo statico per il controllo dei parametri, in modo tale da impedire la costruzione di nodi con informazioni incompatibili con NetLogo.

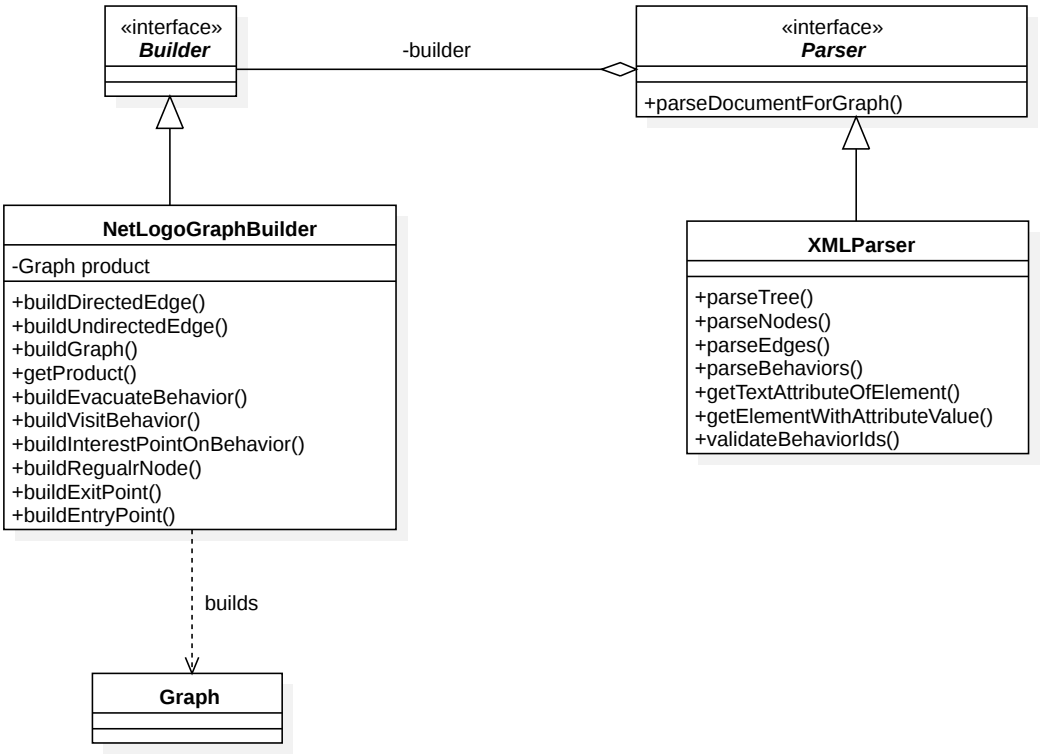


Figura 3.2: Class diagram di Parser e Builder

## 3.2 Builder

Per la costruzione delle classi abbiamo usato il pattern Builder, in modo da mantenere il processo di costruzione della struttura interna dell'oggetto di tipo Grafo trasparente all'utente di questo strumento. In particolare l'interfaccia del Builder aiuta a mantenere l'algoritmo di interpretazione dell'XML

separato dal processo di costruzione e rappresentazione del suo contenuto in classi Java.

Il pattern Builder, inoltre, porta il grande beneficio di migliorare la modularizzazione derivante dall'incapsulamento di tutta la procedura di manipolazione della struttura interna dell'oggetto costruito, in questo modo il Client non ha necessità di conoscere la sua composizione interna.

Una conseguenza che rende il pattern ancora più adatto al nostro caso è il miglioramento del controllo sul processo di costruzione del prodotto. Al contrario di altri pattern creazionali, i quali costruiscono e restituiscono il prodotto in un'unica funzione, il Builder separa queste due operazioni mettendo a disposizione un'interfaccia più completa.

Grazie a questo noi siamo in grado di esercitare un maggiore controllo sulla correttezza delle operazioni eseguite e delle informazioni inserite durante questa prima fase.

L'interfaccia Parser rende il progetto aperto ad estensioni future, come il supporto di linguaggi alternativi all'XML.

In Figura 3.2 si può osservare la struttura delle classi che implementano questo pattern.

### 3.2.1 Funzionamento

La classe `XMLParser` svolge il ruolo di *Director* all'interno del pattern e, durante la visita della struttura JDOM [3] creata, chiama i metodi del *ConcreteBuilder* `NetLogoGraphBuilder` per la costruzione dei vari oggetti e la loro composizione a formare la struttura centrale.

Quando la visita da parte del parser è conclusa viene recuperato il *Product*, che non è altro che un oggetto di tipo `Graph`, in Figura 3.3 si può seguire il processo di costruzione di un semplice modello.

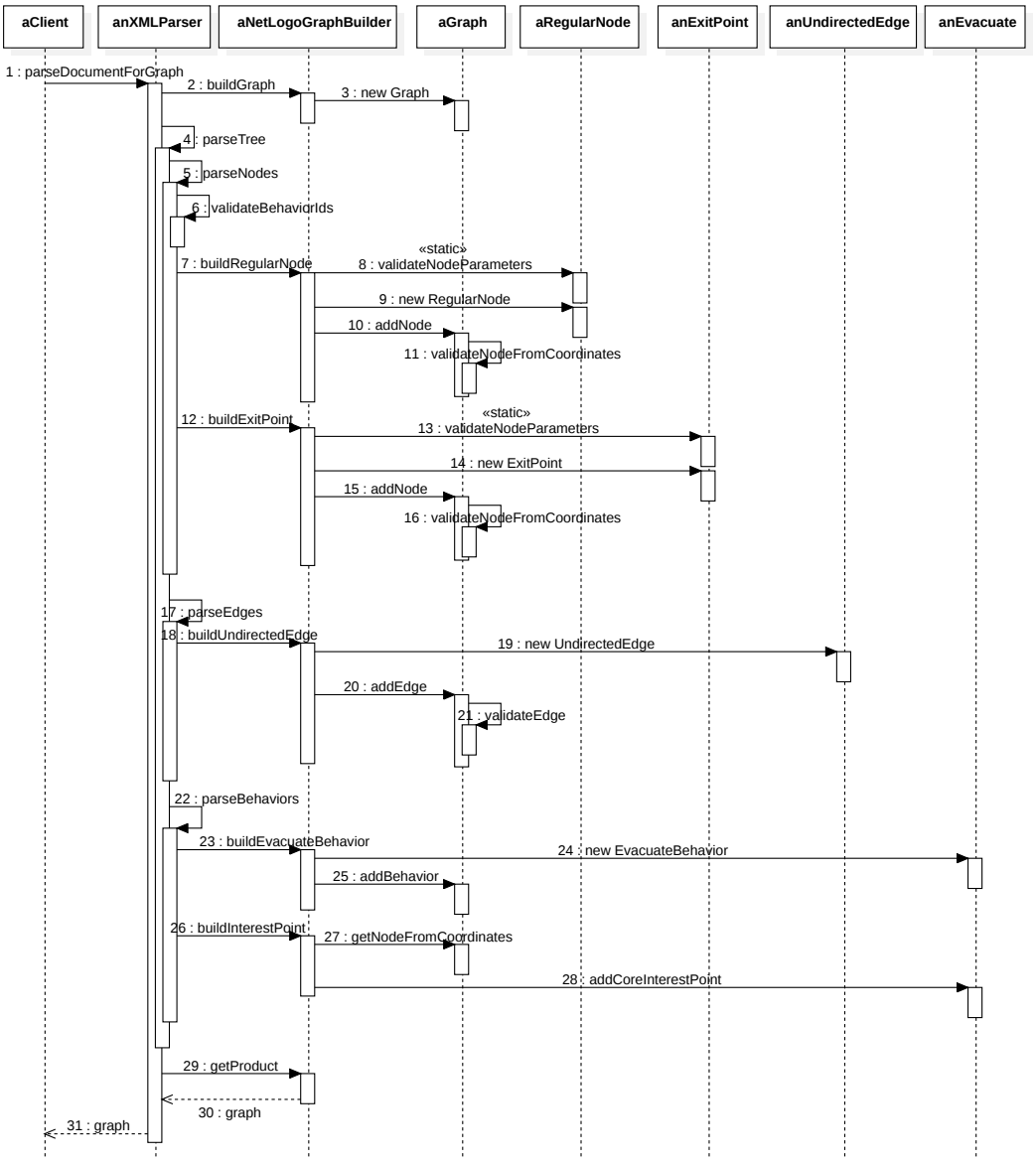


Figura 3.3: Sequence diagram della costruzione di un grafo da parte di XMLParser

Come prima cosa vengono costruite le classi relative ai nodi, `XMLParser` estrae le informazioni dalla struttura JDOM e controlla che non vi siano mancanze, quindi chiama lo specifico metodo di costruzione del builder.

Il `NetLogoGraphBuilder`, prima di procedere alla messa in vita dell'oggetto, verifica, tramite un metodo statico della relativa classe, che le informazioni da inserire siano corrette, ad esempio si controlla che le dimensioni fisiche dell'incrocio non siano negative. Una volta costruito l'oggetto il builder chiama la funzione della classe `Graph` per aggiungere il nodo alla struttura, questa operazione è preceduta da un controllo sui nodi già presenti per evitare sovrapposizioni.

In modo analogo vengono costruiti e aggiunti alla struttura anche gli archi. Si costruisce l'arco e si delega a `Graph` la responsabilità di evitare l'inserimento di grafi con estremi uguali.

Infine si costruiscono i behaviors sui quali poi si aggiungono i riferimenti ai nodi di interesse sfruttando i metodi esposti dalla classe `Graph`.

L'operazione finale eseguita dal `NetLogoGraphBuilder` è quella di restituzione del prodotto costruito tramite il metodo `getProduct()`.

## 3.3 Visitor

Per l'operazione di scrittura del codice NetLogo si ha la necessità di estrapolare dagli oggetti che costituiscono la struttura del grafo diverse informazioni. Al variare della classe concreta varieranno anche le informazioni specifiche da estrarre.

Il grafo inoltre presenta una struttura di classi con interfacce diverse che devono, quindi, essere modificate per poter eseguire l'operazione necessaria.

In questo contesto, quindi, il pattern Visitor si rivela molto utile permettendo di eseguire operazioni specifiche al variare della classe concreta senza inquinare le diverse interfacce all'interno della struttura.

Un ulteriore vantaggio presentato dal Visitor è quello di concentrare le funzioni per la scrittura del codice NetLogo in classi specifiche rendendo il sistema più facile da manutenére ed eventualmente estendere.

La struttura dei modelli NetLogo finali è tale che gran parte del codice necessario rimane fissa al variare della simulazione, per questo motivo si ha che l'operazione di scrittura del codice NetLogo eseguita dai visitor è alternata a quella di lettura da appositi file di testo che contengono la parte fissa del modello. Queste parti fisse quindi vengono completate con le informazioni estrapolate dagli oggetti Java precedentemente messi in vita.

Abbiamo inoltre scelto di separare il modello NetLogo finale in due parti, mantenendo distinta la parte del modello che mette in vita l'ambiente da quella che controlla i movimenti degli attori e raccoglie le informazioni di interesse.

Si hanno quindi due classi visitor distinte per ognuno dei due modelli NetLogo da creare: `NetLogoGraphVisitor` e `NetLogoBehaviorVisitor` ( Figura 3.4 ). Il primo effettua una visitita su nodi e archi del grafo per scrivere i comandi necessari a mettere in vita l'ambiente corretto. Il secondo, invece, effettua una visita sui behaviors in modo da definire i comportamenti che gli attori potranno avere nella simulazione e sui nodi per impostare lo stato iniziale del sistema.

In questo modo le interfacce delle classi del grafo, rimangono inalterate ma le operazioni effettuate variano al variare del visitor concreto, rendendolo un pattern ancora più adatto a questo specifico caso.

NetLogoBehaviorVisitor, in modo analogo, eseguirà operazioni di lettura

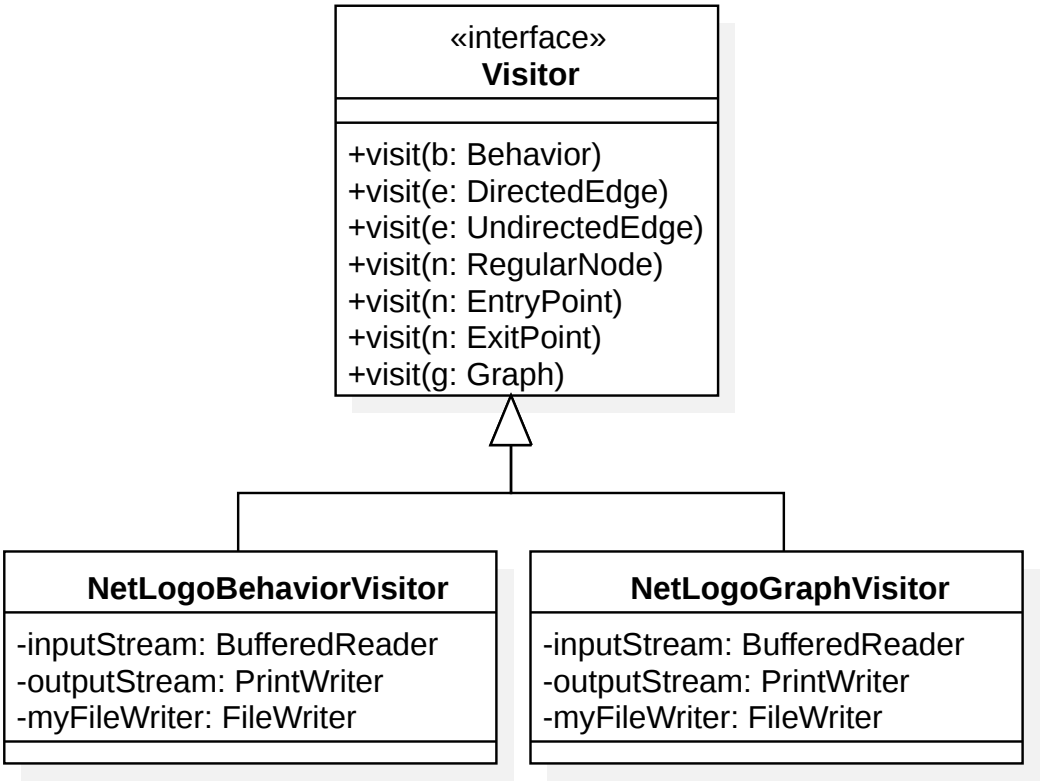


Figura 3.4: Class Diagram del Visitor

e scrittura alternate per generare un un secondo file `.nlogo` in cui si esegue la simulazione e si esportano i dati di interesse.

### 3.3.1 Funzionamento

In Figura 3.5 è mostrato il funzionamento di un `NetLogoGraphVisitor`.Come prima cosa si trascrive una parte del codice contenuto nel file di testo fino alla specifica stringa di flag "`;;===start`".

Quindi vengono estratte dal grafo informazioni riguardanti le dimensioni globali dell'ambiente che deve essere generato e vengono scritte le relative procedure NetLogo per la sua impostazione.

Il visitor estrae dal grafo le liste dei nodi e degli archi che lo costituiscono

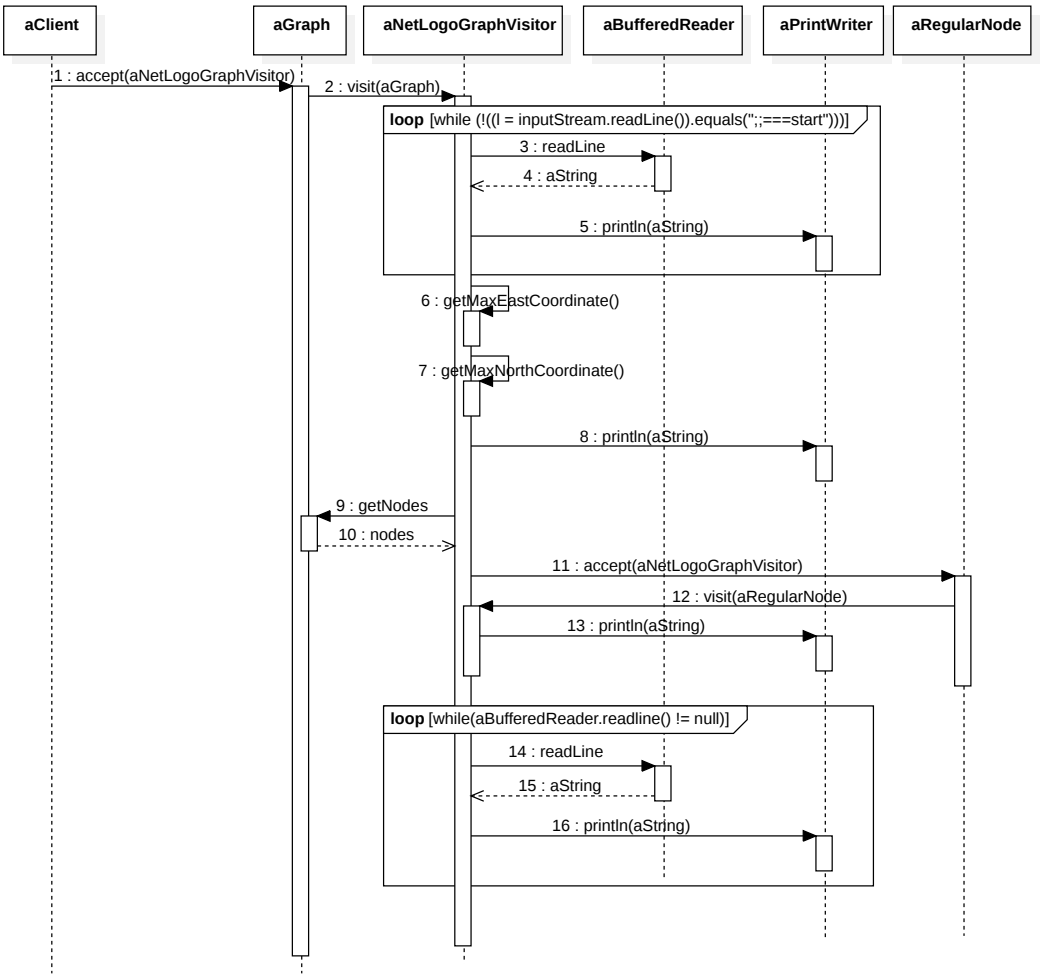


Figura 3.5: Sequence Diagram del NetLogoGraphVisitor su un grafo con un unico nodo

e procede alla loro visita.

Trattandosi del visitor per la generazione dell'ambiente fisico, vengono estratte solo le informazioni riguardanti le dimensioni fisiche delle entità rappresentate dagli oggetti. In particolare per i nodi di ingresso e uscita vengono anche impostate le variabili relative al tasso di generazione o eliminazione e le liste sulle percentuali relative ad ogni behavior.

Si noti che questo visitor non aggiunge alcun tipo di funzionalità alle classi che rappresentano i comportamenti.

In modo analogo `NetLogoBehaviorVisitor` trascrive parte del codice dal file di testo fino ad arrivare alla stringa "`;;===behaviors`".
Il visitor quindi esegue una visita su tutti i behaviors presenti nel modello Java andando a scrivere in NetLogo le procedure per la creazione di una apposita tabella in cui si indicizzano tutti i comportamenti con i loro punti di interesse.

A questo punto si trascrive un'altra porzione di codice dal file di testo fino ad arrivare alla stringa "`;;===system`" che indica il punto all'interno del file in cui devono essere inserite le procedure per l'impostazione dello stato iniziale.

Vengono visitati nuovamente tutti i nodi da cui, ora, si estraggono informazioni sulla quantità di attori presenti e i loro comportamenti. Queste informazioni vengono usate dal visitor per popolare una specifica lista del modello NetLogo in cui si è conservato lo stato.

Anche in questo caso, si può osservare che `NetLogoBehaviorVisitor` non aggiunge funzionalità agli archi del grafo.

## 3.4 Documento XML

La struttura del documento XML prevista per il funzionamento del nostro strumento si attiene il più possibile allo standard di formato GraphML [2] (approvato dal W3C) in modo da evitare inconsistenze e incomprensioni, soprattutto per la parte in cui è descritta la topologia dell'ambiente, per la quale questo formato è pienamente adatto.

Il file XML è suddiviso in tre sezioni distinte:

- `Graph`
- `Behaviors`
- `System`

`Graph` rappresenta la topologia dell'ambiente in cui gli attori si muovono, ed è descritta sotto forma di grafo con `edges` che rappresentano le strade e `nodes` che rappresentano gli incroci.

Gli edges possono essere `directed` e `undirected`, per ognuno di essi vengono specificati peso e larghezza, ovvero `weight` e `eWidth`. Il formato GraphML, inoltre, permette di specificare un tipo di default per gli edges come valore dell'attributo `edgedefault` di Graph.

I nodes invece possono essere di tre tipi: normal, entry o exit. Per tutti e tre i tipi vengono specificate coordinate spaziali e dimensioni fisiche dell'incrocio che esso rappresenta, quindi avremo `nx`, `ny`, `nWidth`, `nHeight` e `nRadius`. All'interno del codice si usa l'attributo `ntype` per specificare il tipo di nodo e anche in questo caso è possibile specificare un tipo di default settato a `normal`.

Per quanto riguarda i `Behaviors` si ha una struttura a grafo in cui ogni nodo rappresenta un behavior di cui viene indicata la tipologia, ovvero `bType`, l'identificatore e la lista dei nodi di interesse.

La sezione `System` descrive lo stato iniziale dell'ambiente. Per questa parte del modello si è preferito distaccarci leggermente dallo standard, in modo da avere una struttura più comprensibile, usando gli specifici tag `<behavior \>`. Per ogni nodo, sotto il tag "`graph`" con identificatore `state`, viene indicato il comportamento e la quantità degli attori presenti. In particolare, nei nodi contrassegnati come entry o exit, vi è un ulteriore tag "`graph`" con identificatore `parameters` in cui si specificano la frequenza di generazione o eliminazione degli attori e le percentuali relative ad ogni behavior. Per i nodi di ingresso viene anche indicato un limite superiore agli attori generabili (`entry-limit`), che può essere messo negativo nel caso in cui, invece, non si vogliano porre limiti.

### 3.4.1 Esempio

Di seguito è riportato un XML in cui si mostra la definizione di ogni tipo di entità coinvolta: attributi, tutti i tipi di nodi, tutti i tipi di archi, behavior e stato associato ai nodi.

```
<?xml version="1.0" encoding="UTF-8"?>
<grafoxml>
  <!-- attributi per i nodi -->
  <key id="nType" for="node" attr.name="type"
     attr.type="string">
    <default>normal</default>
  </key>

                          [...]

  <graph id="Graph" edgedefault="undirected">
    <!-- nodo regolare -->
    <node id="n0">
      <data key="nx">10</data>
      <data key="ny">10</data>
```

```
    <data key="nWidth">5</data>
    <data key="nHeight">5</data>
    <data key="nRadius">3</data>
</node>
```

[...]

```
<node id="n17">
    <!-- nodo di uscita -->
    <data key="nType">exit</data>
    <data key="nx">50</data>
    <data key="ny">10</data>
    <data key="nWidth">5</data>
    <data key="nHeight">5</data>
    <data key="nRadius">3</data>
</node>
<node id="n18">
    <!-- nodo di ingresso -->
    <data key="nType">entry</data>
    <data key="nx">0</data>
    <data key="ny">40</data>
    <data key="nWidth">5</data>
    <data key="nHeight">5</data>
    <data key="nRadius">3</data>
</node>
```

[...]

```
<!-- arco percorribile in entrambi i versi -->
<edge id="e01" source="n0" target="n1">
    <data key="weight">3</data>
    <data key="eWidth">3</data>
</edge>
<!-- arco percorribile in un unico verso -->
<edge id="e12" source="n1" target="n2">
    <data key="eType">directed</data>
    <data key="weight">3</data>
    <data key="eWidth">3</data>
</edge>
```

[...]

```
</graph>

<graph id="Behaviors">
  <!-- comportamento -->
  <node id="b0">
    <data key="bType">visit</data>
    <graph id="0">
      <!-- nodo di intersse -->
      <node id="n3"/>
    </graph>
  </node>
</graph>

<graph id="System">
  <!-- stato di un nodo regolare -->
  <node id="n0">
    <graph id="state">
      <behavior id="0">  <!--  id del behavior -->
        <data key="moverQuantity">50</data>
      </behavior>
    </graph>
  </node>
```

[...]

```
  <node id="n17">
    <!-- parametri di un nodo di usita -->
    <graph id="parameters">
      <data key="rate">0.1</data>
      <behavior id="0">  <!--  id del behavior -->
        <data key="percentage">1</data>
      </behavior>
    </graph>
    <!-- stato del nodo -->
    <graph id="state">
      <behavior id="0">  <!--  id del behavior -->
```

```
          <data key="moverQuantity">0</data>
        </behavior>
      </graph>
    </node>
    <node id="n18">
      <!-- parametri di un nodo di ingresso -->
      <graph id="parameters">
        <data key="rate">0.05</data>
        <data key="entry-limit">50</data>
        <behavior id="0">  <!--  id del behavior -->
          <data key="percentage">1</data>
        </behavior>
      </graph>
      <!-- stato del nodo -->
      <graph id="state">
        <behavior id="0">  <!--  id del behavior -->
          <data key="moverQuantity">0</data>
        </behavior>
      </graph>
    </node>
```

[...]

```
  </graph>
</grafoxml>
```

## 3.5 Parser XML

Come come verrà spiegato nella Sezione **??**, abbiamo scelto il formato JDOM per la rappresentazione del file XML. La scelta di questo formato è stata dettata dalla sua struttura, che si adatta meglio ai nostri fini di sola lettura e non di manipolazione del documento.

La struttura ad albero del modello generato rispecchia il documento XML. Ogni nodo dell'albero è di tipo `Element` e conserva una lista dei figli e un

riferimento al padre, espone inoltre una ricca interfaccia con cui si possono estrarre tutte le informazioni di interesse come gli attributi associati al tag rappresentato, oppure il testo sottostante.

Il parser XML inoltre esegue controlli mirati a verificare che siano fornite tutte le informazioni necessarie all'esecuzione del modello NetLogo, ad esempio controlla che per ogni nodo sia specificato lo stato iniziale, compresi parametri come tasso di generazione o eliminazione per nodi di ingresso o uscita.

## 3.6 Modello NetLogo

Come già accennato in precedenza abbiamo scelto di dividere il modello NetLogo generato dal nostro programma in due parti con lo scopo di mantenere le funzioni necessarie per la costruzione dell'ambiente separate da quelle che modellano i movimenti degli attori e raccolgono le informazioni sulla simulazione eseguita. In questo modo il codice è più facile da gestire e manutenere.

Questa scelta porta, inoltre, il vantaggio di eseguire il primo modello una vola sola e di sfruttare, invece, il file di stato che esso genera nel caso in cui si vogliano simulare le interazioni di diversi comportamenti nello stesso ambiente.

### 3.6.1 Ambiente

L'ambiente in cui gli attori si muoveranno è modellato sotto forma di grafo con specifiche tipologie di turtles dette `beacons` che fanno da nodi e rappresentano quindi gli incroci tra le vie percorribili.

Gli archi sono rappresentati da due tipologie diverse di turtles dette `streets`, percorribili in entrambi i versi, e `directed-streets`, percorribili solo in un verso.

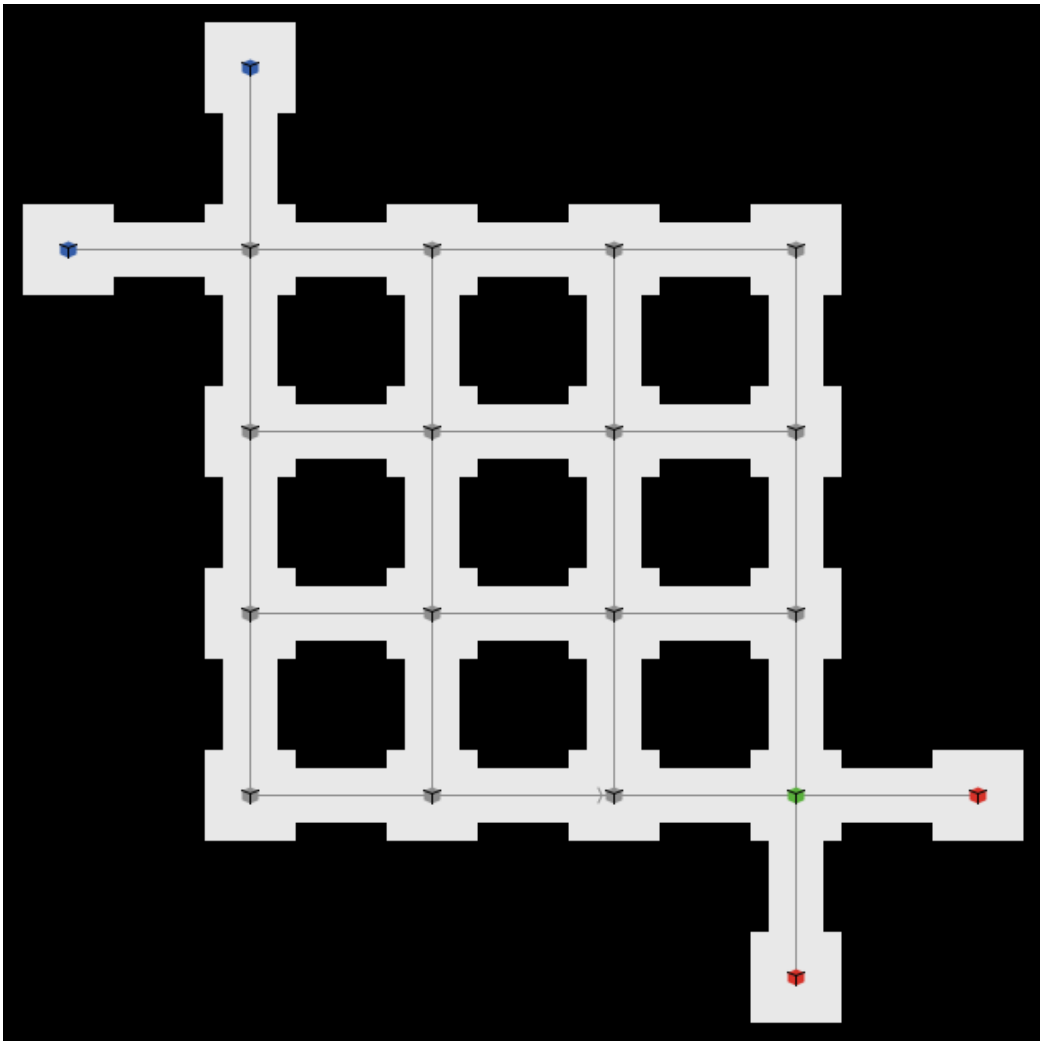

Figura 3.6: Esempio di ambiente generato. In questo caso i beacons di ingresso sono di colore blu e i beacons di uscita di colore rosso.

Per i beacons di ingresso e di uscita vengono anche impostati i relativi parametri, ovvero tasso di generazione o eliminazione degli attori e percentuali relative ai behavior da attribuire ad essi. In particolare per gli entry

points si imposta anche un limite massimo del numero di attori che possono essere generati.

Grazie a questi parametri il sistema è predisposto per lo studio di simulazioni in ambienti comunicanti, in cui quindi gli attori possono uscire da un ambiente e entrare in un altro.

### 3.6.2 Behaviors e stato iniziale

Abbiamo scelto di caratterizzare i behaviors degli attori con un identificatore intero e una lista di nodi di interesse che l'attore deve raggiungere prima di uscire dall'ambiente.

Sono state inoltre inserite due possibili modalità di raggiungimento dei beacons di interesse: "`minDistance`" e "`orderedList`". Nella prima l'attore punta al nodo più vicino a quello precedentemente raggiunto tra quelli di interesse, nella seconda invece la lista viene rigidamente percorsa rispettandone l'ordine.

Lo stato iniziale del sistema è contenuto in una semplice lista `initial-state` che viene percorsa da una apposita funzione per la corretta generazione degli attori sui vari nodi.

## 3.7 Librerie usate

La scelta del linguaggio XML è dettata dalla sua diffusione, con lo scopo di ridurre le conoscenze preliminari necessarie per l'utilizzo di questo strumento e per facilitare la sua integrazione con altri componenti.

Per l'analisi del documento XML esistono diverse alternative, tra le quali le più interessanti sono: DOM, SAX e JDOM.

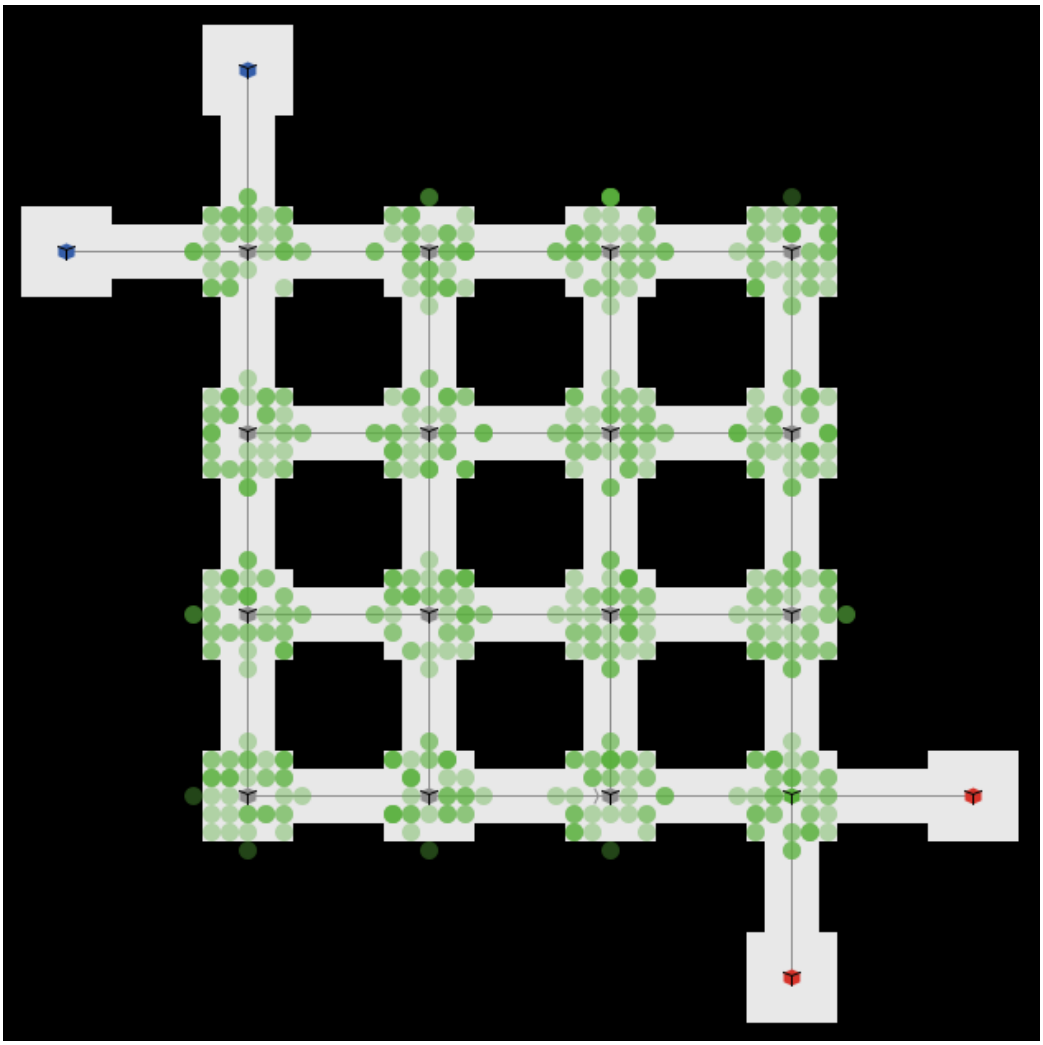

Figura 3.7: Esempio di ambiente popolato con attori nello stato iniziale. Gli attori hanno assunto il colore del beacon che hanno come obiettivo, in questo caso abbiamo un unico behavior che ha come interest point il beacon di colore verde.

DOM (Document Object Model) è uno standard cross-platform e language-indipendent stabilito dal W3C per la rappresentazione di documenti strutturati, quindi XML, HTML, XHTML, come modelli orientati agli oggetti. Il difetto di questo formato e delle sue API, però, sta nella pesantezza in memoria.

La tecnologia SAX (Simple Api for Xml) rappresenta un'alternativa veloce e potente nel caso in cui si voglia gestire il documento con un approccio event-driven.

JDOM [3], infine, è un formato open-source disegnato appositamente per Java che completa DOM e SAX semplificando la manipolazione dei documenti XML. Questo al contrario delle due alternative non è incluso nel JDK, ma è ampiamente accettato dalla comunità.

Il metodo più rapido e semplice per analizzare un documento XML e costruire un modello JDOM è attraverso le API SAX, che JDOM, grazie alla sua natura open source, può sfruttare.

La API JDOM2 associata al formato, quindi, usa la classe `SAXBuilder` [5] per costruire il modello JDOM sotto forma di albero che rispecchia la struttura del documento da cui è stato estratto.

Per quanto riguarda la parte della scrittura del codice NetLogo, si ha che gran parte del codice che modella la simulazione rimane fissa al variare dei modelli descritti negli XML, quindi abbiamo pensato di mantenere questa in un semplice file di testo che viene letto e integrato con le informazioni contenute negli oggetti Java. Per le operazioni di lettura e scrittura su file abbiamo usato le classi `BufferedReader` [1] e `PrintWriter` [4] che permettono lettura e scrittura di intere linee di testo.

Lo strumento segue il seguente workflow:

- analisi del documento XML e costruzione di una rappresentazione della

struttura orientata agli oggetti in JDOM attraverso la relativa API;

- costruzione degli oggetti Java per la rappresentazione della simulazione descritta nel modello JDOM;
- visita degli oggetti Java e scrittura su file del codice NetLogo che eseguirà la simulazione.

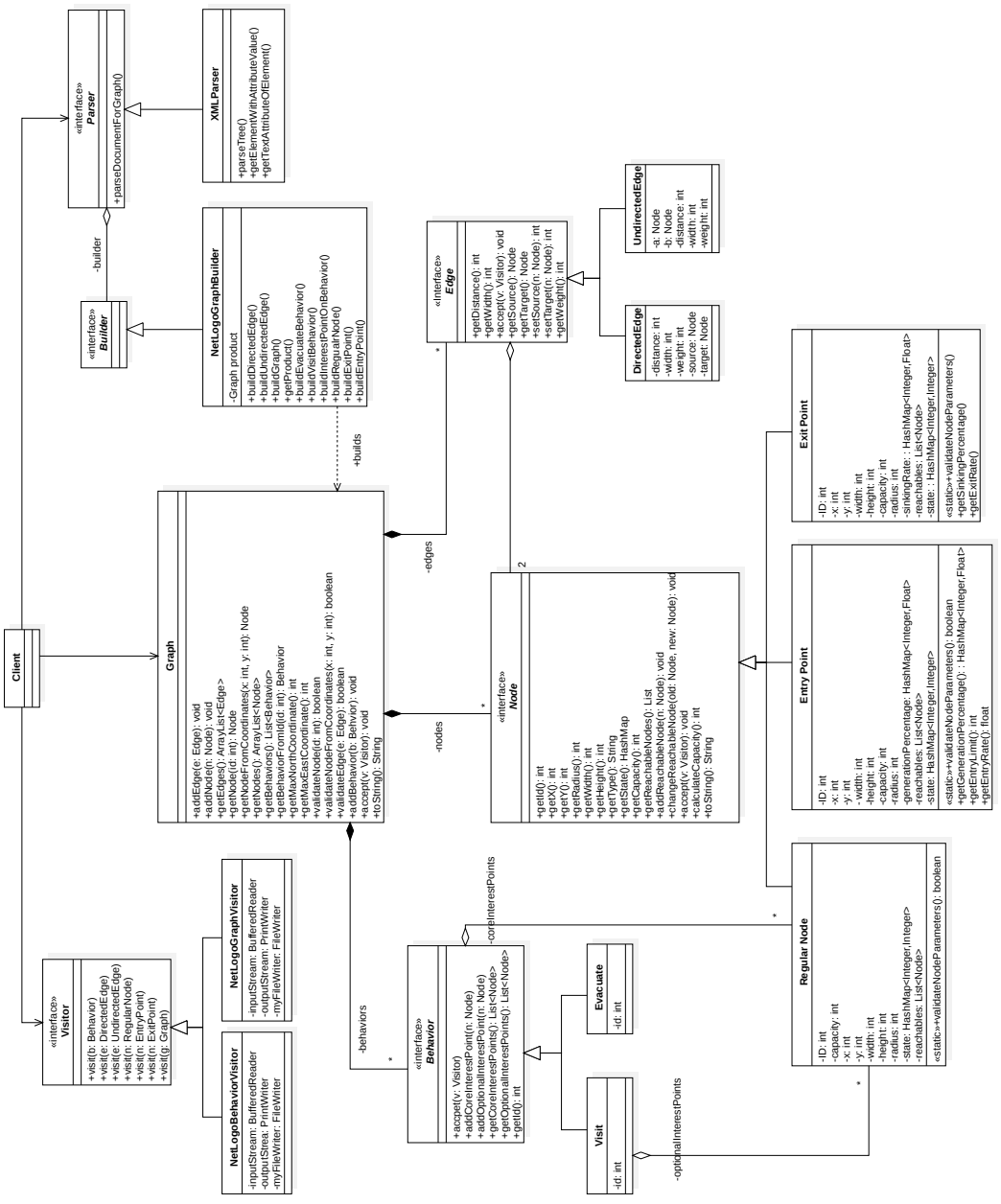


Figura 3.8: Classi diagram

# Capitolo 4

# Esperimenti

Nella fase di sperimentazione del componente ci siamo posti come obiettivo quello di mettere a confronto la simulazione di un modello monolitico e completo con le simulazioni localizzate su specifiche sezioni in cui quest'ultimo è stato diviso. In particolare abbiamo deciso di confrontare il modello completo con due scenari diversi in cui quest'ultimo è stato diviso in 6 e 12 macro regioni adiacenti.
La mappa usata per gli esperimenti è tratta da un blocco residenziale della città di Firenze, in cui abbiamo cercato di inserire diverse topologie in modo da rappresentare zone con strade brevi e strette come il centro storico e strade di dimensioni maggiori come nelle moderne metropoli.

## 4.1 Modello monolitico

La struttura del modello monolitico è rappresentata in figura.

Lo strumento sviluppato in questo progetto di tesi è stato utilizzato dal dottor Sandro Mehic nelle attività di sperimentazione in [6] per la raccolta

dei tempi di soggiorno degli attori nelle singole regioni. Questi dati sono stati utilizzati per il calcolo delle probabilità di transizione delle Catene di Markov che astraggono il comportamento dell'attore attraverso le varie regioni in cui il modello è stato diviso.

dfghjkl

# Bibliografia


[1] BufferedReader. https://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html.

[2] GraphML. http://graphml.graphdrawing.org/.

[3] JDOM. http://www.jdom.org/.

[4] PrintWriter. https://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html.

[5] SaxBuilder. http://www.jdom.org/docs/apidocs/org/jdom2/input/SAXBuilder.html.

[6] Combinig simulation and field analysis in quantitative evaluation of crows evacuation scenarios. 2016.

[7] Jan Dijkstra, Harry JP Timmermans, and AJ Jessurun. A multi-agent cellular automata system for visualising simulated pedestrian activity. *Theory and Practical Issues on Cellular Automata*, 2001.

[8] Dirk Helbing, Ill és Farkas, and Tamas Vicsek. Simulating dynamical features of escape panic. *Nature*, 2000.

[9] Clifford Lasser and Stephen M. Omohundro. *The Essential Star-lisp Manual*. Thinking Machines Corporation.

[10] J-Y Le Boudec, David McDonald, , and Jochen Mundinger. A generic mean field convergence result for systems of interacting objects. In *Quantitative Evaluation of Systems, 2007. QEST 2007. Fourth International Conference*, pages 3–18. IEEE, 2007.

[11] Sandro Mehic, Marco Paolieri, and Enrico Vicario. Research report of hierarchical approach for the scalable analysis of crowd evacuation models. *Università degli Studi di Firenze*, 2015-2016.

[12] S. Okazaki. A study of pedestrian movement in architectural space. part 1: pedestrian movement by the application of magnetic models. *Trans. of AIJ*, 1979.

[13] Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books.

[14] Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. *ACM SIGGRAPH computer graphics*, 1987.

[15] Uri Wilensky. Netlogo (and netlogo user manual). *Center for Connected Learning and Computer-Based Modeling, Northwestern*, 1997. `http://ccl.northwestern.edu/cm/starlogot/`.

[16] Uri Wilensky. StarLogoT. *Center for Connected Learning and Computer-Based Modeling, Northwestern University*, 1997. `http://ccl.northwestern.edu/cm/starlogot/`.

[17] Uri Wilensky and Seth Tisue. Netlogo: Design and implementation of a multi-agent modeling environment. *Agent2004. Chicago, IL. Retrieved October 4, 2012*, 2004.