



SIMULATOR DE TRAFFIC

LUCRARE DE LICENȚĂ

Absolvent: **George-Laurențiu GALIȘ**

Coordonator științific: **Prof. Dr. Ing. Sebestyen-Pal GHEORGHE**

2022



FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

DECAN,
Prof. dr. ing. Liviu MICLEA

DIRECTOR DEPARTAMENT,
Prof. dr. ing. Rodica POTOLEA

Absolvent: **George-Laurențiu GALIȘ**

TITLUL LUCRĂRII DE LICENȚĂ

1. **Enunțul temei:** *Implementarea unei aplicații pentru simularea traficului rutier urban cu scopul optimizării sistemului de semaforizare.*
2. **Conținutul lucrării:** *Introducere, Obiectivele proiectului, Studiu bibliografic, Analiză și fundamentare teoretică, Implementare, Testare și validare, Manual de instalare și utilizare, Concluzii, Bibliografie, Anexa 1.*
3. **Locul documentării:** Universitatea Tehnică din Cluj-Napoca, Departamentul Calculatoare
4. **Consultanți:**
5. **Data emiterii temei:** 1 noiembrie 2021
6. **Data predării:** 8 iulie 2022

Absolvent: _____

Coordonator științific: _____

**FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL CALCULATOARE****Declarație pe propria răspundere privind
autenticitatea lucrării de licență**

Subsemnatul(a) **Galiș George-Laurențiu**, legitimat(ă) cu CI seria XV nr. 223300 CNP 1990616330221 autorul lucrării Simulator de trafic elaborată în vederea susținerii examenului de finalizare a studiilor de licență la Facultatea de Automatică și Calculatoare, Specializarea Tehnologia Informației din cadrul Universității Tehnice din Cluj-Napoca, sesiunea iulie a anului universitar 2021-2022 declar pe propria răspundere că această lucrare este rezultatul propriei activități intelectuale, pe baza cercetărilor mele și pe baza informațiilor obținute din surse care au fost citate, în textul lucrării, și în bibliografie.

Declar că această lucrare nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române și a convențiilor internaționale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în fața unei alte comisii de examen de licență.

În cazul constatării ulterioare a unor declarații false, voi suporta sancțiunile administrative, respectiv, *anularea examenului de licență*.

Data

02.07.2022

Nume, Prenume

Galiș George-Laurențiu

Semnătura

Cuprins

Capitolul 1. Introducere	1
1.1. Contextul proiectului	1
1.2. Motivația	2
Capitolul 2. Obiectivele Proiectului	4
Capitolul 3. Studiu Bibliografic	6
3.1. Definiții	6
3.2. Proiecte implementate	9
3.2.1. SUMO	9
3.2.2. Visual Traffic Simulation	9
3.2.3. Traffic-simulation	10
Capitolul 4. Analiză și Fundamentare Teoretică	12
4.1. Diagrama use-case și structura logică a aplicației	12
4.1.1. Flow-ul aplicației	12
4.1.2. Diagrama use-case	14
4.1.3. Diagramele de activitate	15
4.1.4. Diagramele de secvență	17
4.2. Analiza elementelor de trafic ce trebuie modelate	20
4.3. Citirea și interpretarea datelor de intrare	20
4.4. Modelarea agenților	21
4.5. Algoritmii folosiți pentru calculul variabilelor agenților	22
4.6. Maparea datelor	23
4.7. Afișarea rezultatelor	23
4.8. Suport tehnologic pentru aplicația de simulare	24
4.8.1. Java	24
4.8.2. Java Swing	24
4.8.3. Model-View-Controller(MVC)	25
Capitolul 5. Proiectare de Detaliu și Implementare	26
5.1. Schema generală a aplicației	26
5.2. Proiectarea aplicației	27
5.2.1. Diagrama de pachete	27
5.2.2. Diagrama de clase	28
5.3. Implementarea modulelor	29
5.3.1. Model	29
5.3.1.1. Clasa Vehicle	30
5.3.1.2. Clasa Car	31
5.3.1.3. Clasa Road	31
5.3.1.4. Clasa Crossroad	32
5.3.1.5. Clasa Light	33
5.3.1.6. Clasa StartPoints	34
5.3.2. View	35
5.3.2.1. Clasa Traffic	36

5.3.2.2. Clasa BackgroundMenuBar	39
5.3.2.3. Clasa RoundedJTextField	39
5.3.3. Controller.....	40
5.3.3.1. Clasa Map	41
5.3.3.2. Clasa ReadFromFile.....	42
5.3.3.3. Clasa WriteToFile.....	44
5.3.3.4. Clasa GenerateVehicles	45
5.4. Ordinea evenimentelor din aplicație	46
5.5. Instrumente utilizate.....	46
Capitolul 6. Testare și Validare.....	47
6.1. Cazuri de testare	47
6.2. Rezultate obținute	50
Capitolul 7. Manual de Instalare și Utilizare	53
7.1. Instalare	53
7.2. Interfața grafică	54
7.3. Utilizarea aplicației	55
Capitolul 8. Concluzii	57
8.1. Scurt rezumat și rezultate obținute.	57
8.2. Dezvoltări ulterioare	58
Bibliografie	59
Anexa 1.....	61

Capitolul 1. Introducere

Odată cu apariția digitalizării și a calculatoarelor, multe informații și date din lumea reală pot fi procesate cu ajutorul tehnologiilor, programelor și a algoritmilor pentru producerea anumitor rezultate. De exemplu calcularea rapidă a numerelor, afișarea grafică, crearea de sunete, entertainment, etc., pe scurt toate dispozitivele și funcționalitățile acestora folosite de noi în momentul de față pentru a ne face viața mult mai ușoară, pot fi prelucrate în acest mod.

1.1. Contextul proiectului

Înainte de erea informațională, implementarea fizică în diferite domenii cum ar fi ingineria, medicina, și chiar și arta, venea cu diferite riscuri și minusuri deoarece un produs sau proiect dezvoltat putea fi analizat doar după ce era implementat fizic. Astfel, era destul de dificil de știut cum se va manifesta acesta în timp sau cum va interacționa cu alte obiecte.

Acum avem la dispoziție diferite instrumente pentru rezolvarea acestor probleme, cele prezentate în această lucrare fiind **simulatoarele**. Acestea pot exista în orice domeniu deoarece reprezintă o viziune cât se poate de clară a anumitor scenarii sau obiecte propriu-zise înainte ca acestea să fie implementate în viața reală.

O simulare sau un simulator reprezintă o operație ce „îmită” un proces din lumea reală. Componenta principală ce se află la baza acesteia este un **model** ce reprezintă un obiect sau un comportament în cadrul unui sistem sau proces. Modelul poate produce rezultate sau își poate schimba starea după efectuarea anumitor operații pe parcursul unei perioade de timp stabilite (de exemplu cum evoluează un obiect pe parcursul unei zile) sau nestabilite (de exemplu de cât timp are nevoie un obiect să ajungă din punctul A în punctul B). Modelul este creat pe baza informațiilor ce îl descriu, informații ce pot fi citite din fișier sau introduse de la tastatură.

Unul dintre avantajele simulării este aceea că se creează un **mediu închis** ce poate fi schimbat pentru a testa evoluția și comportamentul obiectelor în diferite scenarii sau în diferite perioade de timp. Cele mai multe simulatoare sunt executate și create pe calculatoare, deoarece, astfel, datele și comportamentul lor pot fi vizualizate într-un mod grafic sugestiv pe un monitor, facilitând manipularea și înțelegerea acestora și în lipsa unei expertize extinse a utilizatorului.

Aceste caracteristici importante ale simulatoarelor rezultă în **reducerea semnificativă a costurilor financiare și temporale ale proiectelor**, deoarece e mult mai ușor, ieftin și rapid să creezi un simulator decât cumpărarea și implementarea componentelor fizice.

Interpretarea datelor din simulator se poate face în orice mod și se poate vedea cât se poate de clar cum au evoluat obiectele pe o durată de timp îndelungată în doar câteva secunde. Acest lucru permite să vizualizăm impactul anilor pe termen scurt, cât și pe termen lung, astfel încât să putem lua decizii ce duc la implementări optime.



1.2. Motivația

Un exemplu de situație, în care cea mai bună practică este dezvoltarea unui simulator înainte de implementarea propriu-zisă, este cea de proiectare și planificare a străzilor și semafoarelor într-un oraș, în vederea decongestionării traficului aferent. Aceasta este și proiectul dezvoltat de mine și prezentat mai jos în această lucrare.

Implementarea și dezvoltarea traficului urban sau rural este de mare prioritate deoarece aceasta este una dintre mai importante caracteristici ce țin de **infrastructură**. Cu cât aglomerația mașinilor într-un oraș este mai mare, cu atât o infrastructură deficitară devine mai aparentă deoarece se produce **ambuteiaj**, fapt ce, implicit, afectează calitatea vieții locuitorilor.

O problemă majoră ce apare odată cu intensificarea traficului este creșterea exponențială a nivelului de poluare a aerului, acel oraș devenind unul „gri”. Cu cât ambuteiajul este mai frecvent și prelungit, mai ales la orele de vârf, cu atât autovehiculele vor degaja mai multe noxe în aer, astfel creând efectul de „smog” – poluare atât de intensă încât reduce vizibilitatea aerului. Acest fenomen este preponderent prezent în marile orașe și metropole ale lumii, în special în capitale, care tind să fie cele mai traversate rutier.

Problema poluării este greu de remediat fără a adresa ineficacitatea infrastructurii, deoarece pentru o curățare oarecum eficientă și pasivă a aerului, un oraș ar trebui să fie proiectat în prealabil cu spații ample de vegetație în contextul în care emisiile de dioxid de carbon nu sunt la un nivel prea ridicat, ceea ce nu este posibil de realizat în majoritatea cazurilor. O altă problemă, ce apare odată cu o infrastructură deficitară, este timpul irosit în trafic. Un ambuteiaj la un moment de vârf al zilei într-un oraș mare poate să rețină deplasarea oamenilor pentru ore întregi. Aceasta perioadă petrecută în trafic rezultă în folosirea inefficientă a timpului șoferilor, la un consum mult mai ridicat de combustibil și, implicit, cheltuieli mai mari pentru aceștia.

O posibilă rezolvare, ce ocolește rădăcina problemei, este implementarea și promovarea unui bun sistem de transport în comun, însă acesta nu poate deservii dorinței de transport independent al unei populații în creștere, care va fi în favoarea deținerii de numeroase vehicule proprii. Contaminarea aerului cu emisii și congestionarea ar putea fi stopate complet doar odată cu utilizarea majoritară de autovehicule electrice - ce nu poluează și prezintă posibilitatea unui trafic programat anticipat pentru o deplasare eficientă - însă privind posibilitățile economice și tehnologice actuale, această soluție e rezervată viitorului. Cea mai fezabilă soluție contemporană, în acest sens, este proiectarea unei infrastructuri bune.

Din perspectiva administrării și dezvoltării unui oraș trebuie testate și măsurate diferite valori esențiale înainte de implementarea infrastructurii în viața reală, de exemplu: perioadele de timp ale mașinilor cât acestea stau la semafor, timpul în care o mașină ajunge din punctul A în punctul B, de câte ori o mașină stă la același semafor.

După măsurarea acestor valori se pot lua diferite decizii ce duc la o **implementare optimă** a traficului. Se poate observa pe ce stradă se produce ambuteiaj și cum sau unde se poate implementa un traseu cu undă verde - mașinile prind culoarea verde la mai multe semafoare consecutiv dacă au o anumită viteză și nu întâlnesc obstacole. Mai mult de atât, pe baza datelor, ce se cunosc după un timp într-un oraș – precum străzile unde se creează un trafic mai intens - se pot seta timpurile semafoarelor pentru a avea un trafic cât mai fluid. De exemplu dacă o stradă nu se aglomerează cu multe mașini, se setează semaforul să țină culoarea roșu mai mult, iar pe altă stradă unde traficul este mai intens, se setează culoarea verde să țină cât mai mult. În acest scop, folosirea unui simulator poate facilita implementarea schimbărilor menționate în prealabil.

În zilele noastre, se dezvoltă diferite simulatoare, ce pot avea ca scopuri secundare educația și entertainment-ul utilizatorilor. Aceste simulatoare au numele de „**serious games**” sau „**jocuri serioase**”. În aceste jocuri, utilizatorul este pus în situația agenților ce vor urma să fie implementați în viața reală. Aceste simulatoare pot învăța oamenii ce să facă în anumite scenarii extreme sau să învețe cum să folosească anumite instrumente. De exemplu, piloții de avioane învață să piloteze în simulatoare de acest gen pentru a putea înțelege fiecare componentă a cabinei de pilotaj. Un alt exemplu este și simularea traficului, în care utilizatorul este pus în situația unui șofer oarecare. Pe baza acțiunilor făcute de utilizator într-un anumit mediu, se pot prelua diferite rezultate și se pot schimba diferite caracteristici din acel mediu pentru a îmbunătăți fluxul aplicației.

Un exemplu de astfel de simulator este prezentat în Figură 1.1. Acest tip de simulator este unul dintre cele mai avantajoase deoarece se bazează pe acțiunile unor oameni în diferite scenarii, iar pentru a genera cât mai multe rezultate se pot crea boți, care să facă acțiuni specifice.



Figură 1.1. Exemplu de „serious game” pentru trafic

Capitolul 2. Obiectivele Proiectului

Așa cum am prezentat în capitolul introductiv, am ales să dezvolt un proiect ce are ca scop simularea traficului unui oraș. Pentru realizarea acestui simulator trebuie detaliate cât mai multe perspective de proiectare, cum ar fi generarea manuală a oricărei tip de hărți, definirea dimensiunilor tronsoanelor, generarea fluxului de mașini, vizualizarea grafică a desfășurării simulării și generarea de rezultate valide.

Așadar, obiectivul general este acela de a crea un simulator, ce permite dezvoltarea diferitelor tipuri de scenarii de trafic și modificarea timpurilor semafoarelor pentru a crea rezultate optime. Pe baza acestor rezultate și scenarii se pot lua decizii pentru executarea proiectului în viața reală.

O aplicație de acest gen trebuie să permită:

- **Desenarea hărților**

În simulatoarele mai moderne sunt implementate și componente grafice ce ajută la vizualizarea comportamentului sau evoluției modelelor/agenților. Vizual se poate observa cel mai bine atunci când ceva este în neregulă. Singurul dezavantaj al vizualizării grafice este dacă simulatorul poate genera o evoluție rezultată de pe o durată mai lungă de timp într-o perioadă scurtă, deoarece mobilitatea agenților va fi prea rapidă pentru observarea evoluției propriu-zise a acestora, fiind studiabile doar rezultatele finale și momentele în care se produc blocaje.

- **Mișcarea mașinilor**

Vizualizarea agenților, în cazul de față - a mașinilor, ajută la o înțelegere cât mai clară a evoluției acestora. Se poate vedea cum se ajunge la diferite situații extreme (unde se creează blocaje sau ce configurație de semafoare este făcută greșit) sau cum se pot crea trasee cu undă verde pentru diferite drumuri principale.

- **Atribuirea timpurilor la fiecare semafor din intersecție**

Pentru a crea un trafic cât mai fluid trebuie optimizate timpurile semafoarelor. Dacă pe o stradă este un trafic intens, timpul de așteptare la semafoare poate fi redus pentru decongestionarea acestuia, în contrast cu timpul de așteptare de pe o stradă mai puțin frecventată, unde acesta poate fi mărit. Acest lucru se poate face dacă se știe deja că pe strada respectivă se află de obicei mai puține mașini decât pe altele.

- **Simularea diferitelor fluxuri**

Definirea fluxurilor reprezintă generarea anumitor agenți la diferite perioade de timp. Acestea duc la rezultatele așteptate sau neașteptate. Se pot crea intenționat fluxuri ce duc la situații extreme. Pe baza acestor fluxuri se pot schimba diferite valori pentru a îmbunătăți evoluția agenților din simulator. De exemplu, dacă pe o stradă generarea mașinilor se produce mai repede, sunt șanse mari ca pe acea stradă să apară un blocaj la un moment dat.

- **Măsurarea diferiților parametri**

Se pot genera rezultate în funcție de timpul ales de utilizator, așadar se poate observa cum evoluează mașinile într-un caz în care este un trafic intens pe o anumită stradă pentru a vedea care este limita maximă a fluxului de mașini pe strada respectivă.

- **Informarea utilizatorului despre anumite evenimente**

Scopul simulatoarelor este acela de a implementa scenarii din viața reală într-o aplicație. Se pot crea diferite scenarii extreme pentru a se studia comportamentul obiectelor în acel mediu. Prin urmare, este foarte important de observat atunci când se produc anumite evenimente și cum se produc, de exemplu, dacă mașinile stau prea mult la semafor sau când se creează un blocaj pe o anumită stradă.

- **Schimbarea timpului și adaptarea acestuia la scenarii din simulator**

Timpul în simulatoare poate fi interpretat diferit. De aceea se poate considera că o secundă din viața reală nu este aceeași ca o secundă dintr-un simulator. Astfel, pe baza datelor introduse în simulator, se poate vedea ce se întâmplă mai exact în timp real sau pe o perioadă mai lungă de timp, într-un posibil viitor.

- **Proiectarea diagramelor pentru diferitele scenarii de utilizare**

Atunci când se face dezvoltarea unei aplicații de acest gen, una dintre principalele faze de proiectare este implementarea diagramelor pentru cazurile de utilizare. Acestea sunt: diagrama use-case, diagrama de activitate și diagrama de secvență. Pentru fiecare diagramă se analizează acțiunile destinate utilizatorilor și pe baza acestor activități se implementează aplicația propriu-zisă. Aceasta este prima etapă a dezvoltării aplicațiilor, indiferent de aplicație.

Capitolul 3. Studiu Bibliografic

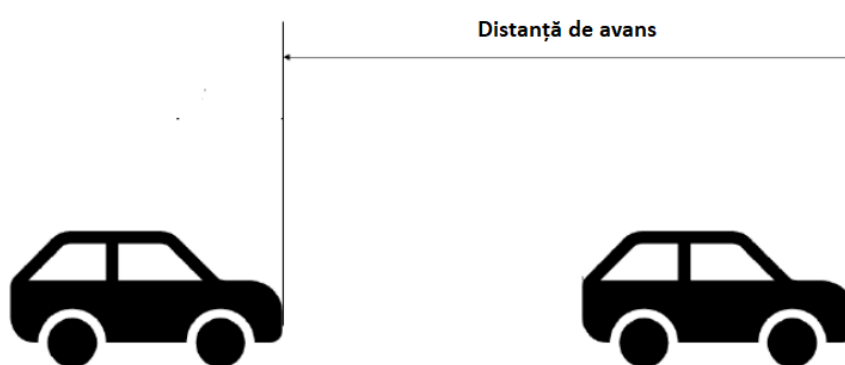
În acest capitol se va face un studiu la referințele din Bibliografie, așadar, se vor lua concepte de implementare și studii făcute în legătură cu simularea traficului, dar și idei spre dezvoltarea unei astfel de aplicații.

În articolul [1] autorul tezei de master prezintă fluxul de trafic eterogen. Fluxul de trafic eterogen poate fi definit ca numărul de vehicule care trec pe lângă un anumit punct, într-un anumit timp, pe un drum cu mai multe mașini. Prin urmare, se dorește realizarea unui trafic cât mai fluid, într-un flux în care este o rată mare de vehicule ce trec printr-un punct [2]. Aceste durate de așteptare duc la dezavantaje de mediu, economice și emoționale, cum ar fi creșterea emisiilor vehiculelor, consumul crescut de combustibil, timpul pierdut și furia rutieră [3].

3.1. Definiții

Pentru a putea compara acest studiu cu alte asemenea studii în ceea ce privește procedurile de testare, rezultatele simulării și concluziile, este important să existe definiții coerente ale măsurătorilor relevante ale performanței de conducere. Acest lucru este realizat cu ajutorul formulărilor date de SAE International în raportul lor, numit „Definiții operaționale ale măsurilor și statisticilor de performanță în conducere” [4]. Bineînțeles, la dezvoltarea aplicației nu trebuie folosite toate conceptele prezentate, doar cele necesare. Câteva dintre definițiile prezente mai jos au termenii în engleză deoarece nu exista termeni specifici în română.

Timpul de avans reprezintă „intervalul de timp dintre două vehicule care trec de un punct măsurat de la bara de protecție din față a unui vehicul la bara de protecție din față a următorului vehicul succesiv” [5] și **distanța de avans** este definită în același mod ca „distanța între bara de protecție din față a unui vehicul și bara de protecție din față a vehiculului precedent” [5]. Un exemplu este ilustrat în Figură 3.1. Calculul acestor valori poate diferi de la aplicație la aplicație. Din punctul meu de vedere, această valoare se poate calcula mai ușor și într-un mod diferit (de exemplu valoarea distanței dintre cele două mașini – bara din spate a unei mașini și bara din față a mașinii succesive).



Figură 3.1. Vizualizare a distanței de avans

Viteza vehiculului și timpul de călătorie, adică timpul necesar unui anumit vehicul pentru a călători de la punctul A la punctul B, este o măsurare importantă a fluxului de trafic. Viteza vehiculului este invers proporțională cu timpul de călătorie sau, cu alte cuvinte, viteza medie mai mare a vehiculului este direct proporțională cu scăderea timpului de călătorie [6]. **Pierderea timpului de călătorie** este diferența dintre timpul de călătorie dorit, adică timpul necesar pentru a parcurge o anumită distanță cu limita de viteză afișată pentru drumul respectiv fără nicio întrerupere, și timpul călătoriei în sine [7]. Exemple de întreruperi care cauzează pierderi de timp sunt **formarea de cozi și întârzierea la plecare**, care este definită ca întârzierea plecării vehiculului din cauza lipsei de spațiu rutier disponibil [8]. Aceste valori, ce trebuie calculate, duc la producerea rezultatelor finale și analiza acestora pentru îmbunătățirea perioadelor de timp în așteptare.

Time-to-Collision și **Vehicle Clearance** – primul termen reprezintă timpul necesar pentru coliziunea a două vehicule dacă continuă cu viteza lor actuală și pe aceeași cale [8]. Aceasta valoare este o măsurare eficientă de evaluare a siguranței traficului și este adesea folosită pentru a identifica și determina severitatea unui conflict de trafic. **Vehicle Clearance** denumește distanța liberă a vehiculului. O reprezentare grafică se poate observa în Figură 3.2.

Error! Reference source not found.



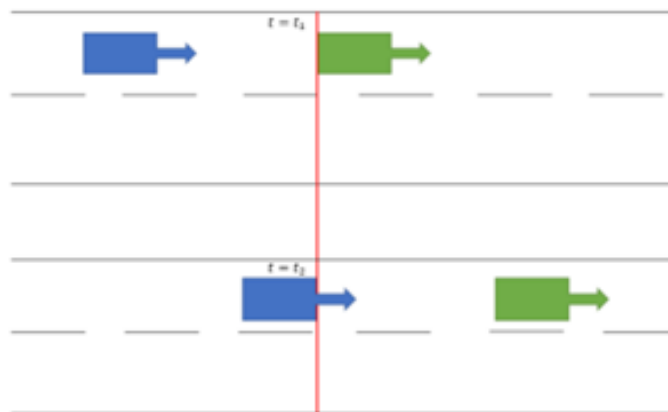
Figură 3.2. Vizualizare a distanței libere a vehiculului

Avantajele aflării acestor valori este acela că pe baza lor se pot modifica date din interiorul simulării pentru un flux cât mai fluid de mașini. Având aceste informații se poate măsura **Time-to-Collision** cu ajutorul formulei (3.1).

$$TTC = \frac{D}{V_e - V_t} \quad (3.1)$$

Post Encroachment Time reprezintă diferența dintre momentul în care un vehicul intră într-un punct de conflict până când un alt vehicul ajunge în acest punct [8]. După părerea mea, această valoare trebuie calculată doar în anumite situații. Formula de calcul al acestui timp este (3.2) (t_2 și t_1 pot fi vizualizate în Figură 3.3):

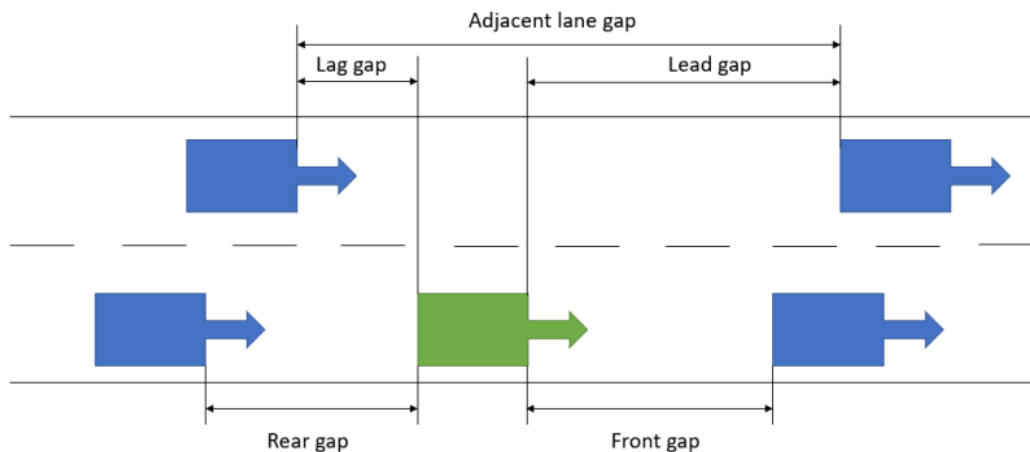
$$PET = t_2 - t_1 \quad (3.2)$$



Figură 3.3. Vizualizare Post Enchroachment Time

Schimbarea benzii reprezintă „mișcarea unui vehicul de pe banda sa inițială, pe o altă bandă, cu deplasarea continuă în aceeași direcție” [8]. În opinia mea, luarea în calcul a acestui termen se poate face doar în cazul în care se dorește o dezvoltare de simulator mai complexă, în care tronsoanele au benzi multiple și agenții pot avea decizia de schimbare a benzii (mai multe calcule).

Descrierea decalajelor în trafic(gaps). Două dintre principalele măsurători care sunt utilizate pentru a determina dacă se produce o schimbare de bandă efectuată în siguranță sunt: decalajul din urmă(lag gap) și decalajul de avans(lead gap). După cum indică măsurătorile din Figură 3.4., decalajul este distanța dintre vehiculul verde și vehiculul de urmărire(albastru) pe banda adiacentă. Avantajele principale ale acestei distanțe este că agentul știe exact când este și când nu este posibilă o schimbare de bandă.



Figură 3.4. Descrierea decalajelor

După cum indică descrierea parametrilor de mai sus, există o legătură puternică între siguranța traficului și fluxul de trafic și, prin urmare, siguranța trebuie luată în considerare atunci când se investighează fluxul de trafic. [6]

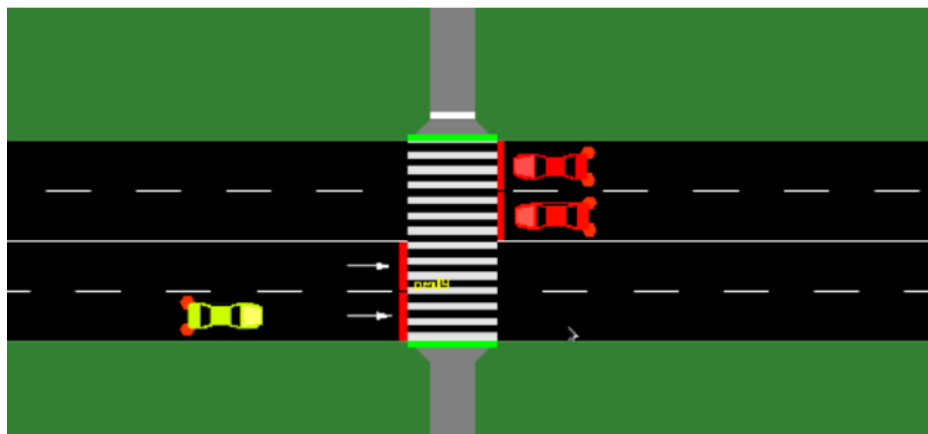
3.2. Proiecte implementate

În documentația [7], se prezintă o aplicație ce simulează traficul în 2D. Cu ajutorul acestei documentații am putut să îmi fac o idee clară despre cum trebuie să arate un astfel de simulator.

3.2.1. SUMO

SUMO a fost dezvoltat de Centrul Aerospațial German (DLR) de la Institutul Sistemului de Transport cu intenția de a fi folosit ca instrument de ajutor atunci când se investighează diferite probleme de trafic, cum ar fi: alegerea rutei și comunicarea vehiculului - să fie generate manual cu aplicația „netgen” sau prin importul digital al unei rețele rutiere reale din exemplu Open Street Map [7].

SUMO este utilizat pentru pregătirea și efectuarea diferitelor tipuri de simulări microscopice, cum ar fi urmărirea mașinilor, schimbarea benzii și intersecțiile. Fiecare rută a vehiculelor și timpul departamentului sunt definite individual, chiar mai mult, parametrii vehiculului, precum viteza, proprietățile fizice și distanța față de celelalte vehicule, pot fi, de asemenea, definite. În Figură 3.5 este o reprezentare grafică a aplicației SUMO.

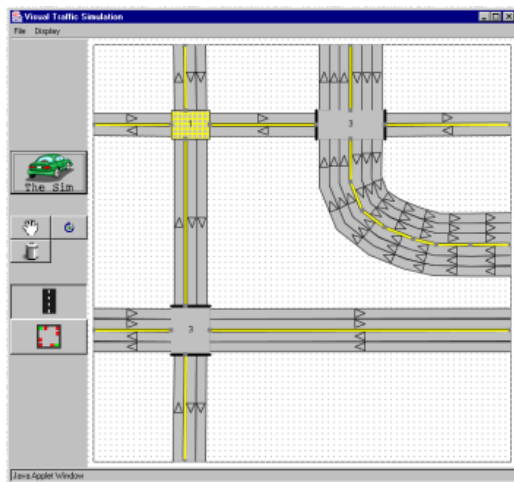


Figură 3.5. Vizualizare a interfeței SUMO

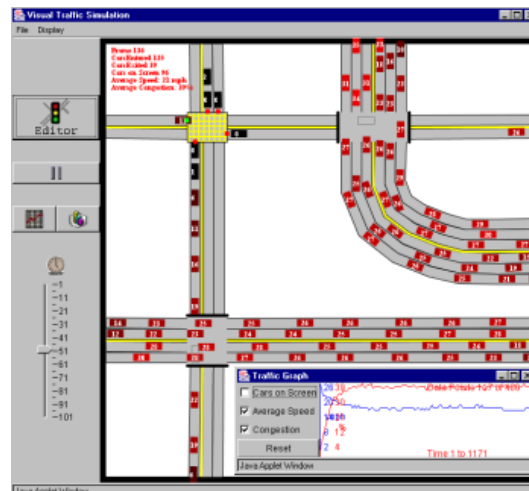
În figura de mai sus este reprezentat un drum construit manual cu o trecere de pietoni și trei modele de vehicule. Simularea este în timp-discret și spațiu-continuu, iar ieșirile de simulare pot fi generate după fiecare pas de timp.

3.2.2. Visual Traffic Simulation

O altă aplicație implementată care a fost de ajutor este prezentată în [8]. Această aplicație este o lucrare de licență de la Imperial College. Aplicația, la fel ca cea prezentată mai sus m-a ajutat să îmi fac o idee despre cum ar trebui să arate o astfel de aplicație. Pe site-ul referențiat la [8] se poate descărca codul aplicației și tot acolo se află și documentația proiectului. După părerea mea, fiind o aplicație destul de veche și neactualizată, nu se mai poate ține cont de ea în zilele noastre, dar pe mine m-a ajutat să îmi fac o imagine asupra bazelor dezvoltării aplicației proprii. Interfața grafică a acestei aplicații este prezentată în Figură 3.6 și Figură 3.67.



Figură 3.6. Vizualizare editor



Figură 3.7. Vizualizare simulare

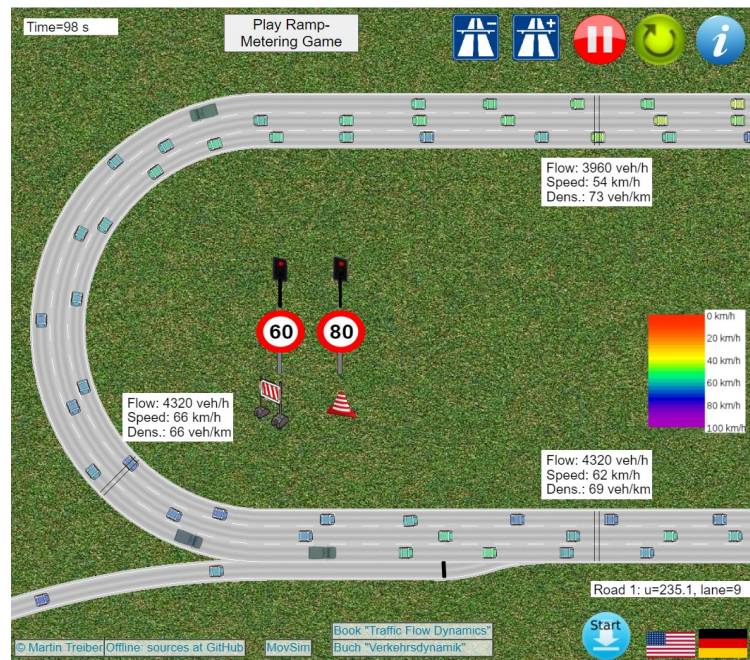
În prima figură se poate observa cum se modelează mediul în care vor evolua agenții. Crearea hărții este procesul de trasare a drumurilor și plasarea de intersecții pentru a forma o infrastructură validă. [8] Ulterior, se poate seta numărul de benzi de pe drumuri, algoritmi pentru semafoare și frecvențele de pornire pentru utilizatorii drumului. Modificările utilizatorilor drumurilor și semaforului nu sunt vizibile în Editor, ci doar în Simulator.

În a doua figură se poate observa cum se desfășoară aplicația și cum evoluează agenții în timp. Simulatorul este locul în care are loc explorarea reală a interacțiunii din trafic. În fereastra Simulator, harta încărcată este animată cu agenți, iar semafoarele arată starea acestora. Singurele modificări care pot fi aplicate în Simulator sunt cele referitoare la viteza mașinilor din sistem, totuși există acces la statisticile pe care simulatorul le generează pe măsură ce rulează. [8]

Cu ajutorul acestei aplicații mi-a venit ideea să implementez o metodă în care simularea se desfășoară pe mai multe trepte de timp. Acest lucru este foarte important deoarece utilizatorul nu trebuie să aștepte până la terminarea simulării și poate crea diferite scenarii pentru o perioadă lungă de timp.

3.2.3. Traffic-simulation

Aceasta este prima aplicație descoperită când am început documentația pentru aplicația proprie. Această aplicație oferă multe caracteristici pentru crearea unei situații de trafic, indiferent de obstacole, număr de străzi, intensitatea fluxului și altele. Această aplicație nu are o documentație vastă, existând acces doar la codul sursă încărcat pe github.com. Din această aplicație am luat ideea de generare de viteză medie în funcție de fluxul de mașini. În Figură 3.8 se poate observa interfața grafică a aplicației, iar link-ul către aplicație se află în Bibliografie [9].



Figură 3.8 traffic-simulation

Capitolul 4. Analiză și Fundamentare Teoretică

În acest capitol se va prezenta tot ce este de înțeles pentru dezvoltarea unei astfel de aplicații. De obicei, un simulator este o aplicație desktop ce se rulează cu ajutorul unui executabil. Aceste aplicații pot fi puse și într-un mediu online, dar un simulator trebuie să proceseze și să ofere rezultate cât mai repede și mai precis. De aceea, majoritatea simulatoarelor sunt aplicații desktop deoarece acestea trebuie să folosească direct resursele calculatorului. Ce se descrie în capitolul următor reprezintă ce ar trebui să știe cineva pentru dezvoltarea unei aplicații asemănătoare și se vor prezenta diferite concepte ce trebuie alese în vederea construirii acesteia. Exemplele de cod pentru termenii respectivi sunt prezentate și referențiate în Anexa 1.

Înainte de dezvoltarea unei astfel de aplicații, sunt câteva criterii ce trebuie luate în considerare, precum: ce paradigme, ce limbaje și ce algoritmi de programare se folosesc, desenarea diagramelor ce descriu arhitectura aplicației și ce decizii se iau pentru funcționarea cât mai optimă a acesteia.

Acestea sunt explicate mai jos.

4.1. Diagrama use-case și structura logică a aplicației

Urmează să prezint diagramele ce caracterizează acțiunile utilizatorului și cum ar trebui să se comporte o aplicație de tip simulator. Aceste diagrame fac implementarea aplicației mult mai ușoară, deoarece se împarte dezvoltarea acesteia în multiple task-uri ce trebuie îndeplinite. Mai mult de atât, se pot adăuga sau șterge funcționalități pe parcursul dezvoltării aplicației.

4.1.1. Flow-ul aplicației

Înainte de a vedea aplicația finală și implementarea propriu-zisă a acesteia, trebuie înțeles care sunt pașii prin care trece procesul de simulare. Flow-ul aplicației este prezentat sub forma unei organigrame în Figură 4.1. Am ales vizualizarea sub forma unei **organigrame** deoarece se poate vedea cel mai clar top-down care sunt pașii esențiali ai aplicației. Aceștia sunt descriși în căsuțele dreptunghiulare, iar deciziile sunt scrise în interiorul romburilor. În funcție de situația întâlnită (dacă răspunsul la întrebarea din romb este „DA” sau „NU”), se decide care este următorul pas. Pot fi mai multe decizii luate în același timp, așa cum se poate observa la pașii „Mașina întâlnește o altă mașină”, „Mașina întâlnește un semafor” și „Semaforul este verde”.

Așadar, primul pas este pornirea aplicației. Etapele pentru pornirea aplicației sunt prezentate în Capitolul 7. După deschiderea aplicației, utilizatorul introduce numele fișierelor de unde se iau datele (se presupune că fișierele JSON sunt scrise deja). Aplicația va genera harta în funcție de datele din fișierele JSON alese de către utilizator. Când utilizatorul pornește simularea, aplicația începe numărătoarea inversă (timpul de simulare se citește tot din fișierul JSON).

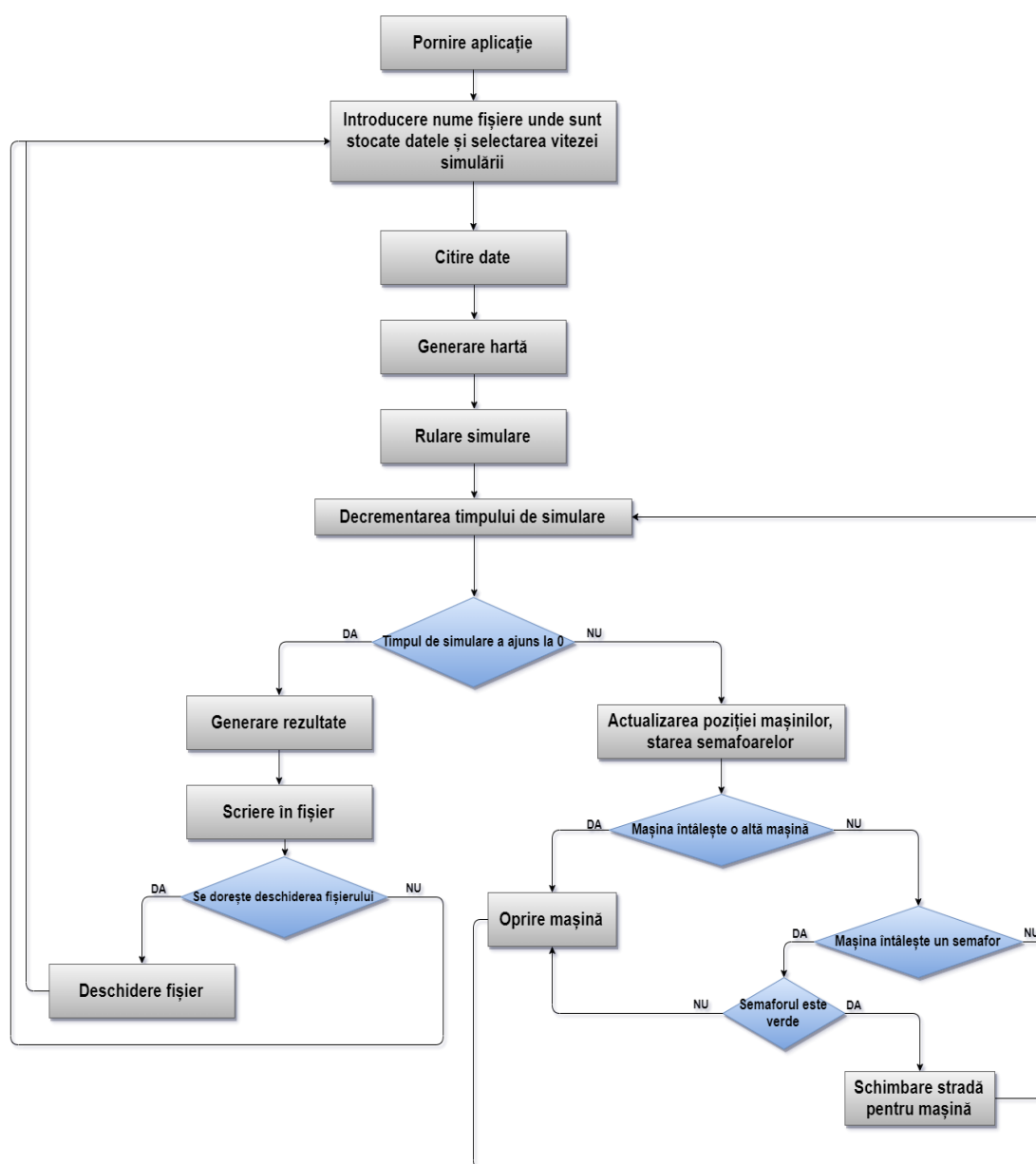
Pașii ce se iau în timpul simulării sunt următorii: se decrementează timpul declarat de utilizator și la fiecare pas de simulare se verifică dacă timpul de simulare a ajuns la 0.

În cazul în care acesta a ajuns la 0, simularea se termină, aplicația generează rezultatele (le mapează la viața reală), le scrie în fișier și întreabă utilizatorul dacă dorește deschiderea și vizualizarea fișierului. După efectuarea acestor pași, utilizatorul

poate rula din nou aceeași simulare sau introduce alte nume de fișiere pentru a rula alte scenarii de simulare.

În cazul în care timpul de simulare NU a ajuns la 0, se actualizează pozițiile mașinilor și starea semafoarelor. La fiecare pas de simulare se verifică dacă mașina inițială (se parcurg toate mașinile și se compară cu celelalte) întâlnește o altă mașină. În caz afirmativ, mașina inițială se oprește. În caz negativ, se verifică dacă mașina întâlnește un semafor. Atunci când întâlnește un semafor există două cazuri: dacă semaforul este verde, mașina va schimba strada pe care aceasta era, iar dacă semaforul este roșu, mașina se oprește și așteaptă ca acesta să se facă verde. Urmatorul pas este decrementarea timpului de simulare și reparcurgerea pașilor de mai sus până când se termină timpul de simulare.

Mai jos sunt reprezentați pașii explicați mai sus sub o formă mult mai logică și mai ușor de înțeles.



Figură 4.1. Flow-ul aplicației

4.1.2. Diagrama use-case

Diagrama use-case prezintă acțiunile făcute de utilizat în cadrul aplicației. Așa cum am prezentat în Capitolul 4, sarcinile, pe care le are de făcut utilizatorul, sunt asemănătoare cu cele prezentate pentru o aplicație de tip simulator generală, numai cu mai puține use-case-uri, deoarece este o aplicație dezvoltată de o singură persoană.

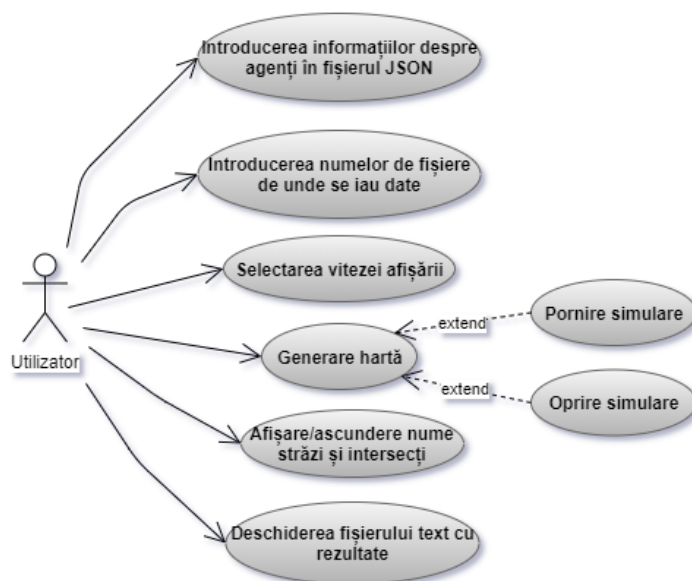
Diagrama use-case este prezentată în Figură 4.2.

După cum se poate observa, sunt puține cazuri de utilizare deoarece un simulator nu se bazează pe acțiunea utilizatorului ca în alte aplicații (de exemplu jocurile video). Acesta are de introdus informații, vizualizează cum evoluează agenții și analizează rezultatele oferite de simulator. Cea mai importantă parte și de ținut cont când se dezvoltă o aplicație de tip simulator este partea efectivă de simulare, nu cea care ține de acțiunile utilizatorului.

Așadar, la partea de **introducere a informațiilor despre agenți**, acesta trebuie să creeze sau să folosească fișierele JSON din directorul „json/maps” sau „json/flows”(aplicația vine cu câteva fișiere deja scrise - Capitolul 7). Introducerea datelor este foarte ușoară, deoarece, în fișierele JSON, datele sunt scrise sub forma cheie:valoare, utilizatorul trebuie doar să respecte formatul JSON și să scrie corect cheile datelor. În cazul în care nu este introdus ceva corect, programul afișează în consolă unde este problema prin excepțiile din clasele folosite pentru citirea fișierelor JSON.

În continuare, utilizatorul va deschide aplicația, se va afișa interfața grafică, iar următorul lucru pe care îl va face utilizatorul este acela de a **introduce numele fișierelor JSON** descrise mai sus, deoarece aplicația va lua datele și va forma mediul simulării.

Utilizatorul va **genera harta** prin apăsarea unui buton, iar în cazul în care ceva nu este în regulă cu datele introduse, nu se va genera harta. Odată ce harta este generată, utilizatorul poate să **selecteze viteza desfășurării simulării** cu ajutorul unui combo box, **va putea porni/ opri simularea în orice moment, va putea afișa/ ascunde numele străzilor, intersecțiilor și direcțiile viitoare ale mașinilor**. Toate aceste acțiuni pot fi făcute în timpul simulării.

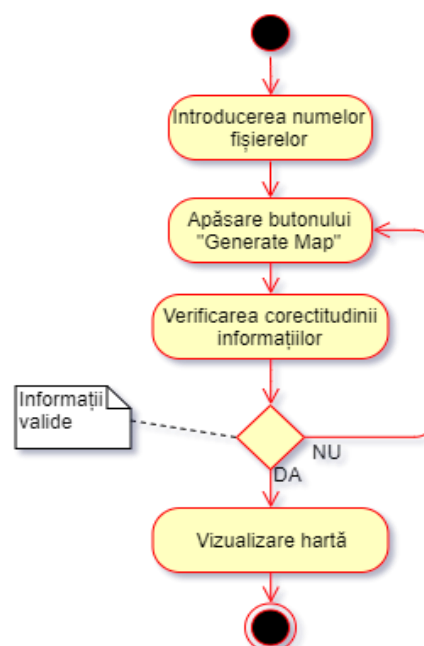


Figură 4.2. Diagrama use-case

4.1.3. Diagramele de activitate

Diagrama de activitate descrie o succesiune de acțiuni(activități) pentru fiecare caz de utilizare. Diagramele de activitate oferă suport pentru comportamentul condiționat(reprezentat de romburi), cât și pentru comportamentul paralel(fork sau bifurcare reprezentat de o linie îngroșată). Începutul „activității” este reprezentat de un punct negru simplu și finalul acesteia de un punct negru încercuit. Diagramele de activitate oferă o viziune mai clară a situațiilor ce trebuie îndeplinite de utilizator pentru fiecare caz de utilizare prezentat mai sus. Astfel, sunt cinci diagrame de activitate, câte una pentru fiecare din următoarele cazuri de utilizare:

La partea de introducere a numelor de fișiere(diagrama prezentată în Figură 4.3), utilizatorul are la dispoziție două text field-uri în care are de introdus numele fișierelor JSON. După ce se introduc datele, se apasă pe butonul „Generate Map” pentru a afișa harta descrisă în fișierele JSON. După verificarea datelor de către utilizator sau de aplicație(aceasta afișează informații în cazul în care nu se găsește fișierul sau nu s-au introdus datele), aplicația va genera harta ce poate fi vizualizată de utilizator și va confirma utilizatorului când totul este în regulă.



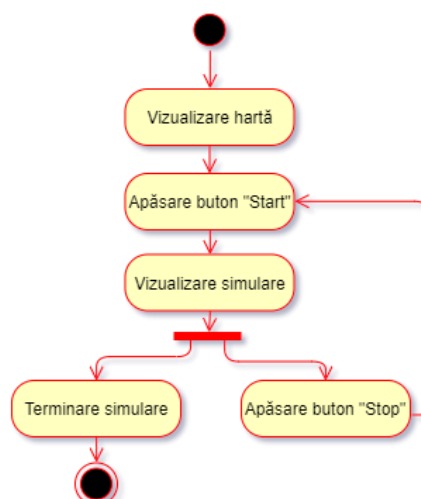
Figură 4.3. Introducere a numelor de fișiere

După ce se generează harta, chiar și înainte să se introducă numele fișierelor, utilizatorul poate să selecteze dintr-un combo box viteza cu care să se desfășoare simularea. Acesta are de ales dintre opțiunile: „Slow”(mașinile și timpul trec mai greu), „Normal”(viteza normală), „Fast”, „Super Fast”(mașinile și timpul se mișcă foarte repede) și „Get Results”(în cazul în care utilizatorul nu vrea să observe cum decurge simularea , ci doar să vadă direct rezultatele scrise în fișierul text). Diagrama de activitate pentru acest caz este ilustrată în figura din dreapta(Figură 4.4).



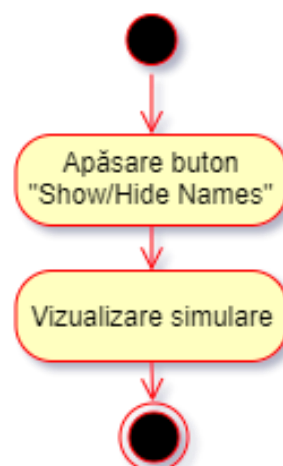
Figură 4.4. Selectare viteză

Utilizatorul se află în cazul în care simularea este pregătită pentru execuție. Odată ce totul este în regulă și numele fișierelor au fost introduse corect, utilizatorul poate să înceapă sau să pună pe pauză simularea în orice moment cu ajutorul a două butoane: „Start” și „Stop”. Odată ce se pune pe pauză simularea, mașinile, starea semafoarelor și numărătoarea inversă se vor opri (mai exact, nu se va intra în if-ul în care rulează thread-ul - detaliile de implementare sunt explicate în 5.3). După ce numărătoarea inversă ajunge la 0, simularea se va termina. Diagrama de secvență a acestui caz de utilizare se poate vizualiza în Figură 4.5.



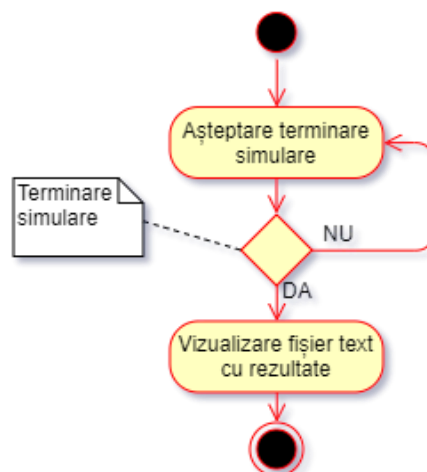
Figură 4.5. Pornire/oprire simulare

Aplicația oferă utilizatorului varianta de a vizualiza/ascunde numele străzilor și intersecțiilor. Mai mult de atât, se poate observa și următoarea direcție a fiecărei mașini în parte (în dreptul fiecărei mașini va fi scris cu roșu „R”(right), „L”(left), „U”(up), „D”(down)). Acest lucru se poate realiza atât în timpul simulării, dar și când simulatorul este oprit. Am ales implementarea unei astfel de funcționalități deoarece mi s-a părut de ajutor atunci când se creează o hartă. Așa se poate observa numele fiecărei intersecții la care e atribuit fiecare drum și astfel se face legătura dintre acestea mult mai ușoară. Diagrama de secvență a acestui caz de utilizare se poate vizualiza în Figură 4.6.



Figură 4.6. Afișare/ascundere nume strazi, etc.

După ce se termină simularea (timpul afișat în simulator a ajuns la 0), aplicația va informa utilizatorul despre acest lucru printr-un mesaj pop up. În acest mesaj, utilizatorul va avea de ales între deschiderea fișierului text, în care sunt stocate rezultatele făcute de simulator, sau închiderea ferestrei de pop up. Am ales ca fișierele să fie stocate într-un fișier text deoarece acesta poate fi deschis pe orice tip de sistem de operare și pentru a putea fi folosite acele date rezultate în alte scopuri, cum ar fi: pentru generare de grafice sau pentru o rețea neuronală. Diagrama de secvență a acestui caz de utilizare se poate vizualiza în Figură 4.7.

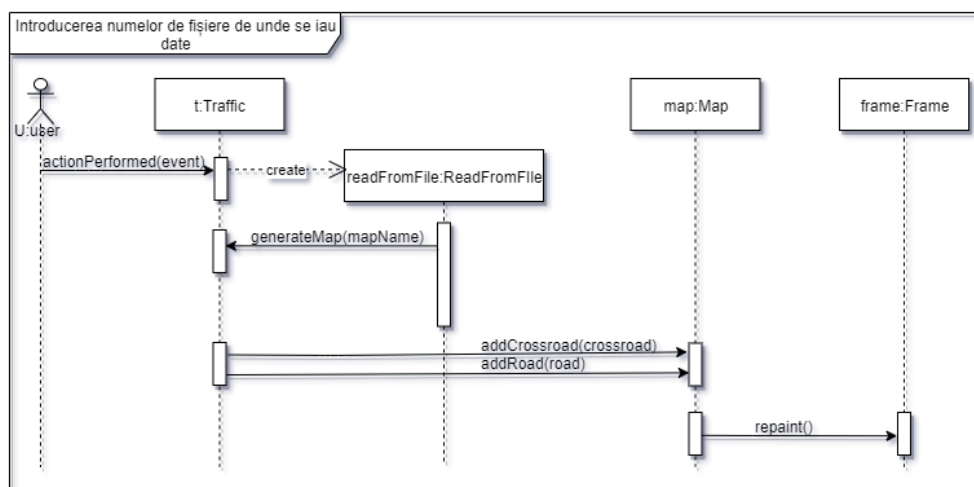


Figură 4.7. Vizualizare rezultate

4.1.4. Diagramele de secvență

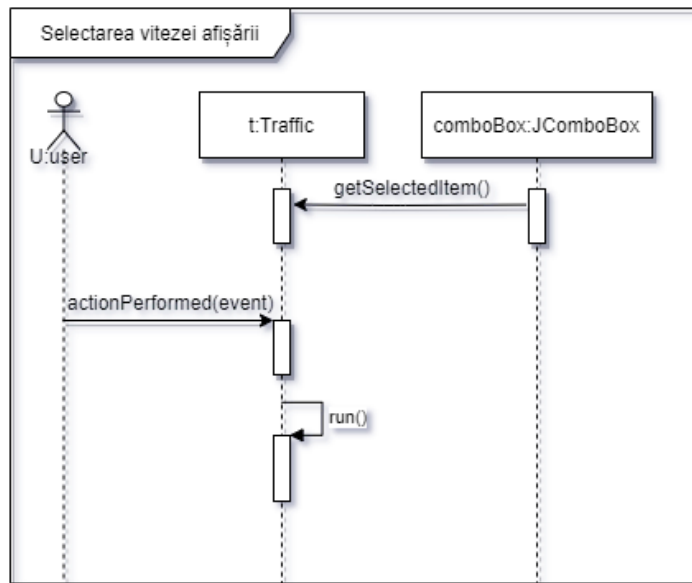
Diagrama de secvență detaliază modul în care sunt efectuate procesele aplicației, surprinzând interacțiunea dintre obiecte [18]. Numele obiectelor este scris în dreptunghiuri sub forma numeleObiectului:numeleClasei. Săgețile reprezintă interacțiunea dintre obiecte și pe acestea sunt scrise numele metodelor ce se folosesc din cadrul obiectelor. Cu ajutorul diagramelor de secvență se poate observa ordinea interacțiunilor și cum evoluează un anumit caz de utilizare în timp. Așadar, am construit câte o diagramă de secvență pentru fiecare caz de utilizare. Utilizatorul este reprezentat de omulețul din stânga care începe diferite acțiuni. Clasele, obiectele și metodele vor fi explicate în detaliu în capitolele 5.2.22 și 5.33. Acum, ne vom concentra doar pe numele acestora și când sunt apelate.

În cazul de utilizare a introducerii numelor de fișiere și de generare a hărții, utilizatorul apasă butonul de generare hartă. Când se întâmplă acest lucru, se apelează metoda din cadrul butonului (ce face parte din librăria wing). Când se apasă butonul, se creează un obiect ReadFromFile, care face citirea din fișierele introduse de utilizator. După citirea datelor din fișier, se generează harta (adică se populează structurile de date din clasa Traffic, ce duc la afișarea grafică a hărții) prin intermediul metodei generateMap(mapName). Odată populate structurile de date cu valorile necesare pentru crearea hărții, acestea vor fi folosite și în clasa Map prin intermediul metodelor addCrossroad(crossroad) și addRoad(road). Când sunt pregătite toate valorile pentru afișarea hărții, se apelează metoda ce face afișarea pe ecran, adică metoda repaint()(metodă din cadrul librăriei Swing). Diagrama de secvență a acestui caz de utilizare este prezentată în Figură 4.8.



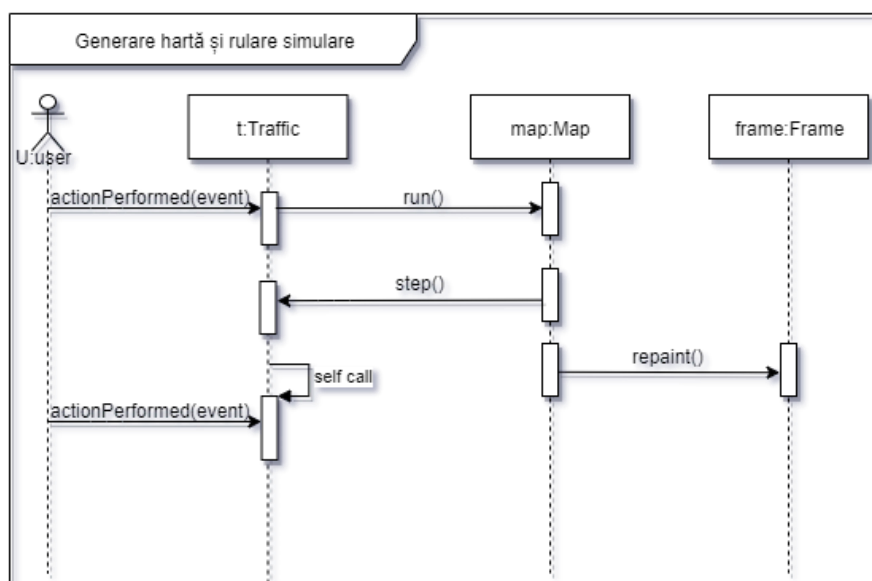
Figură 4.8. Introducere a numelor de fișiere

În cazul selectării vitezei de simulare, utilizatorul alege din componenta declarată din cadrul librăriei Swing, adică JoptionPane, viteza dorită. Ca și la butoane, aplicația trebuie să știe când s-a ales o anumită viteză acest lucru realizându-se cu ajutorul metodei actionPerformed(event). Când se apasă componenta de selectare a vitezei, urmează să se apeleze și metoda getSelectedItem() ce va lua varianta aleasă de utilizator. O dată ce s-a ales varianta dorită, sw va schimba viteza de afișare a simulării și simularea va continua prin metoda run() din cadrul thread-ului ce produce animația. Diagrama de secvență se poate observa în Figură 4.9.



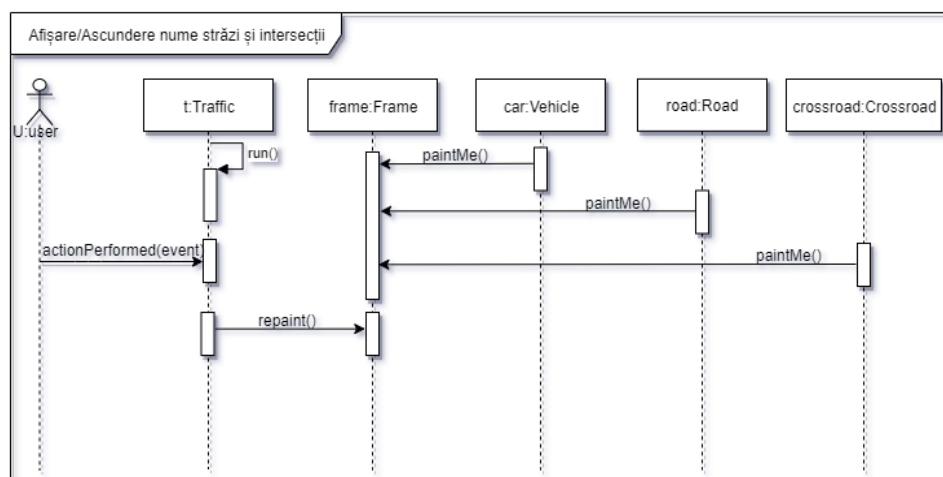
Figură 4.9. Selectare viteză

După introducerea valorilor și generarea hărții, utilizatorul are la dispoziție două butoane, unul pentru începere simulare („Start”) și altul pentru a pune simularea pe pauză („Stop”). Fiecare buton are câte o metodă caracteristică(`actionPerformed(event)`), și în funcție de butonul apăsat, aplicația face acțiunea respectivă. Când se apasă butonul de „Start” se va intra într-un `if` din metoda `run()` ce creează mișcarea. Când se apasă butonul de „Stop” nu se va intra în `if`-ul respectiv și simularea nu va mai crea mișcare. La fiecare apelare a metodei `run()` se apelează metoda din cadrul clasei `Map`, adică `step()` ce actualizează datele fiecărui agent(mașini, semafoare, timp). Acest lucru se întâmplă la fiecare pas de simulare. După ce se actualizează datele din cadrul simulării, se folosește metoda `repaint` pentru afișarea pe monitor a actualizărilor făcute din metoda `step()`. Diagrama de secvență este reprezentată la Figură 4.10.



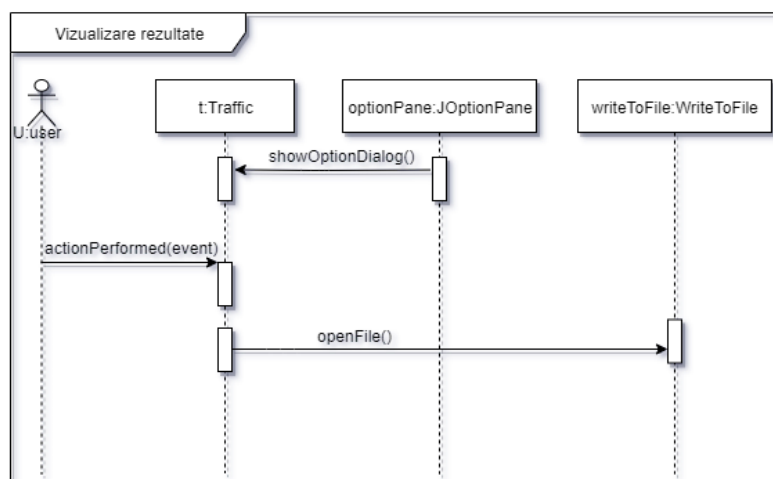
Figură 4.10. Pornire/oprire simulare

Utilizatorul are opțiunea de a ascunde sau a afișa numele intersecțiilor, numele străzilor și următoarea direcție a mașinilor. Această acțiune se face cu ajutorul unui buton de tip toggle. Mai exact, când nu se afișează numele, butonul se numește „Show Names” și atunci când sunt afișate numele, butonul se numește „Hide Names”. Aplicația „știe” când s-a apăsă buton cu ajutorul metodei `actionPerformed(event)`. Acest caz de utilizare se întâmplă în timpul simulării, de aceea se apelează metoda `run()` din clasa `Traffic` (self call). La fiecare apăsare a butonului, se iau numele pentru fiecare componentă în parte (`Vehicle`, `Road`, `Crossroad`) și prin metoda ce afișează pe componentele `paintMe()` se afișează sau nu numele acestora. După ce se actualizează stările componentelor, se apelează metoda `repaint()` pentru afișarea pe ecran. Diagrama poate fi vizualizată în Figură 4.11.



Figură 4.11. Afișare/ascundere nume străzi, etc.

Ultimul caz de utilizare ce îl are de făcut utilizatorul, este cel de a vizualiza rezultatele scrise în fișierul text. După terminarea simulării, aplicația afișează un pop-up prin intermediul clasei `JOptionPane` în care se afișează două butoane: „OK” și „Open File”. Când utilizatorul alege să deschidă fișierul text, se apelează metoda statică din cadrul clasei `WriteToFile` și se deschide fișierul „Results.txt”. Diagrama este prezentată mai jos (Figură 4.12).



Figură 4.12. Vizualizare rezultate

4.2. Analiza elementelor de trafic ce trebuie modelate

Mai sus s-au prezentat concepte și un exemplu de limbaj de programare ce poate duce la dezvoltarea unui simulator într-un mod cât mai ușor și intuitiv. Pe lângă conceptele prezentate mai sus, trebuiesc luate în vedere și alte idei pentru dezvoltarea unei aplicații optime. În cazul unei aplicații de simulare a traficului trebuie luate în considerare cazuri legate de implementare cum ar fi:

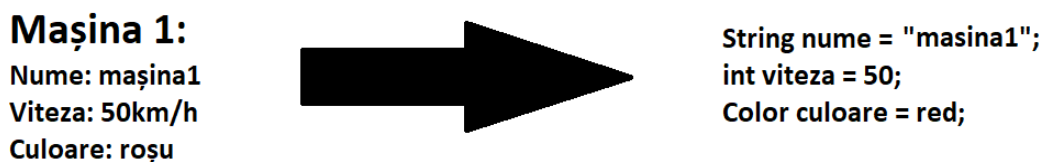
- de unde și cum iau datele ce duc la afișarea mediului în care se dezvoltă agenți;
- cum interpretez datele de intrare;
- ce arhitectură pentru dezvoltarea aplicației folosesc;
- cum modelez mediul;
- cum modelez agenții;
- cum se produce mișcarea agenților;
- ce algoritmi folosesc;
- cum interpretez timpul în simulator;
- cum modelez semafoarele;
- cum fac ca aplicația să fie cât mai intuitivă și ușor de utilizat(user-friendly);
- ce funcționalități sunt destinate userului;
- cum adaptez datele din viața reală pentru mediul creat în simulator;
- cum generez rezultatele în funcție de mediul creat;
- cum generez rezultate pentru a putea fi cât mai ușor de înțeles;
- cum poate fi îmbunătățită aplicația pe viitor;

4.3. Citirea și interpretarea datelor de intrare

Citirea datelor de intrare se poate face în mai multe moduri, alegerea fiind la cel care implementează aplicația. Datele pot fi citite direct de la tastatură prin introducerea lor în text field-uri. Un dezavantaj al acestei metode este acela că datele trebuie introduse de fiecare dată când se rulează aplicația pentru testarea diferitelor scenarii. Un alt dezavantaj este acela că dezvoltatorul trebuie să dezvolte aplicația astfel încât să informeze utilizatorul atunci când s-au introdus date greșite(nu respectă tipul de date cerut, sunt simboluri ce nu sunt înțelese de aplicație, formatul datelor nu este corespunzător). Datele mai pot fi citite cu ajutorul componentelor deja implementate de librăriile folosite de dezvoltator. De exemplu se pot folosi check box-uri, radio box-uri, butoane pentru diferite acțiuni(butoane pentru start/stop de exemplu), combo box-uri în care utilizatorul să aleagă dintr-o listă de opțiuni și multe altele. O alternativă pentru citirea datelor este prin scrierea în fișier a acestora, după care aplicația să interpreteze datele din fișierele respective. Așadar nu trebuie introduse manual valori la fiecare rulare cum ar fi numărul de mașini, coordonatele tronsoanelor sau ale mașinilor, vitezele pentru fiecare mașini, direcții, etc. Acestea pot fi scrise sau modificate direct din fișier, urmând în aplicație să se selecteze fișierul dorit prin introducerea numelui fișierului sau folosind tool-uri implementate deja pentru selecția de fisere(file choosers). Singurul dezavantaj al acestei metode este acela că aplicația trebuie să verifice dacă datele citite respectă un anumit format. În caz că datele sunt introduse greșit, aplicația poate informa utilizatorul unde se află greșeala mai exact(de exemplu să se afișeze în consolă: „Date introduse greșit la linia 10”). Se poate face un program care să genereze date de intrare aleatoare pentru a observa dezvoltarea aplicației în diferite scenarii.

În timpul dezvoltării aplicației, o **practică foarte bună** este să se observe comportamentul agenților în fiecare situație pentru a nu fi detectate la finalul dezvoltării aplicației și să se ajungă să se modifice multe dintre componentele acesteia. Așadar, testările sunt esențiale în timpul dezvoltării aplicației pentru a observa în ce direcție evoluează aplicația. În acest mod se pot detecta și bug-uri ale agenților sau afișări necorespunzătoare a mediului.

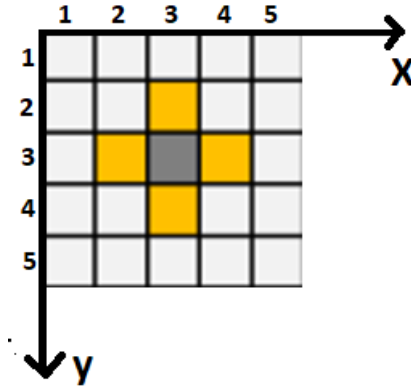
Interpretarea datelor se face la nivelul aplicației. Datele din fișier pot avea un anumit format(de exemplu plain text) la care se poate face o conversie la datele declarate în program. Un exemplu vizual se poate observa în Figură 4.13. Singurul lucru de care trebuie să țină cont dezvoltatorul este acela de cum sunt stocate datele în fișier pentru a ști cum se mapează acestea la nivelul limbajului de programare folosind diferite metode(extragere date din șiruri de caractere, ce date sunt pe o linie din fișier, ce delimitări sunt între date, etc.).



Figură 4.13. Exemplu de conversie

4.4. Modelarea agenților

Modelarea agenților se poate face de către dezvoltator sau se pot folosi medii de tip agent care sunt deja implementați. Aceste medii pot fi folosiți pentru a crea orice tip de agent dar pot apărea probleme de implementare în procesul de dezvoltare a aplicației, de exemplu agenții nu sunt compatibili cu mediul deja dezvoltat, agenții au implementate mai multe funcționalități decât se cere pentru aplicație ducând la încetinirea aplicației, etc. Există cinci tipuri de agenți în funcție de gradul lor de inteligență și capacitate percepută [10]. Dacă dezvoltatorul nu dorește să apeleze la acești agenți deja implementați, acesta poate crea la nivel de aplicație un agent propriu și să fie adaptat specific pentru cerințele aplicației. Așadar, un agent poate fi o formă simplă geometrică ce face diferite acțiuni. De exemplu se desenează un dreptunghi ce reprezintă un camion și acesta își schimbă poziția în funcție de cum sunt declarate tronsoanele. Acest tip de implementare a agenților are un mare avantaj deoarece se adaptează strict la mediul implementat de dezvoltator. Mai mult de atât, evoluția acestor agenți poate fi monitorizată foarte ușor deoarece dezvoltatorul știe cum sunt implementați aceștia și se poate intui care va fi starea următoare a acestora. Astfel, agenții se pot adapta la valorile din simulator cum ar fi timpul și distanțele declarate. Un exemplu de declarare a mediului și al agenților, într-un plan 2D este prin împărțirea panel-ului în valori de coordonate X și Y. Astfel se pot declara modele în funcție de aceste coordonate și producerea mișcării agenților prin incrementarea valorilor X și Y. Un exemplu de ilustrare este prezentat în Figură 4.14.



Figură 4.14. Exemplu de mapare a agenților

4.5. Algoritmii folosiți pentru calculul variabilelor agenților

Algoritmii sunt folosiți în aplicație sunt pentru maparea datelor din viața reală în mediul aplicației. Aceștia sunt folosiți pentru realizarea rezultatelor. Pentru a face posibil acest lucru se folosesc formule pentru calcularea vitezei, distanței și timpului în cadrul aplicației, dar și în cadrul lumii reale. În cazul unui simulator, formulele de bază pentru calculul valorilor sunt următoarele: calculul vitezei (4.1), calculul timpului (4.2) și calculul distanței(4.3).

$$viteza = \frac{distanța}{timp} \quad (4.1)$$

$$timp = \frac{distanța}{viteza} \quad (4.2)$$

$$distanța = viteza * timp \quad (4.3)$$

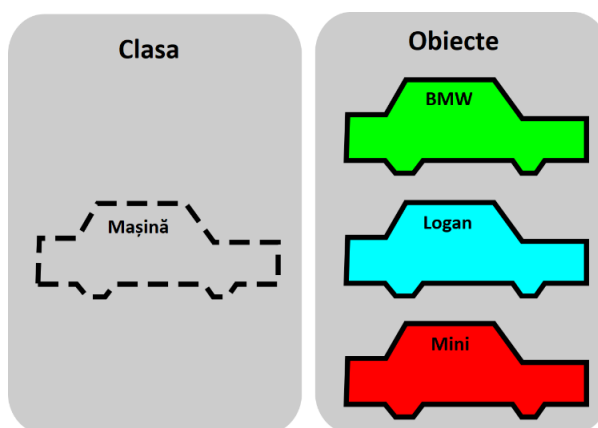
Distanța dintre două puncte este o caracteristică importantă pentru un simulator deoarece știind această distanță se poate monitoriza mai ușor agenții și evoluția acestora. Calcularea distanței se poate face la finalul simulării sau în timpul simulării la fiecare pas de simulare. Știind această distanță constant, se pot pune condiții ce descriu comportamentul agenților sau se pot detecta anumite evenimente, de exemplu coliziunea a două mașini din trafic. Formula pentru calcularea distanței dintre două puncte este următoarea (4.4): [11]

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (4.4)$$

4.6. Maparea datelor

Maparea datelor se face de către dezvoltator deoarece datele diferă de la un simulator la altul. De exemplu un metru din viața reală este diferit de un metru din simulator (acesta poate însemna o valoare prestabilită sau un număr de pixeli). Așadar, datele citite se mapează pentru a putea fi procesate pentru aplicație și generarea rezultatelor, după care se mapează din nou pentru a produce rezultate pentru viața reală.

Mai mult de atât, agenți se pot reprezenta sub formă de clase și obiecte cu ajutorul programării orientate pe obiect. După cum se poate observa în Figură 4.15 orice agent (de exemplu mașină) poate avea o clasă ce reprezintă „rama” (adică caracteristicile unui obiect) și fiecare agent de tipul mașină să poată fie reprezentat de câte un obiect în parte.



Figură 4.15. Reprezentare agent sub formă de obiect

4.7. Afișarea rezultatelor

Afișarea rezultatelor se poate face în multe moduri. Acestea se pot genera la finalul sau în timpul simulării. Rezultatele pot fi afișate direct în simulator în interfața grafică sau într-un document text. Datele rezultate se pot genera pentru diferite scopuri cum ar fi: observarea evoluției agenților din simulare, crearea de grafice, folosirea lor pentru realizarea de predicții folosind o rețea neuronală sau compararea acestora cu rezultate din altă tranșă de simulare.

Simulatoarele generează rezultate în funcție de datele introduse. În majoritatea cazurilor, simulatoarele generează rezultate în funcție de ce alege user-ul. Rezultatele se pot genera în timp real adică în momentul rulării simulării sau la finalul acesteia. Graficele ajută mult la observarea comportamentului agenților în timpul evoluției acestora în simulator. Exemple de grafice ce se pot realiza sunt:

- Viteza medie în funcție de densitate
- Viteza medie în funcție de un număr strict de vehicule

Thread-urile pot fi o componentă principală pentru dezvoltarea unui simulator. Acestea permit programului să opereze mai eficient deoarece se execută mai multe task-uri în același timp. Aceștia pot fi folosiți pentru calcule ce au loc în background fără să întrerupă procesul principal [12]. În simulatoare este foarte important un astfel de concept deoarece agenții trebuie să evolueze independent. Însă, pentru un mediu în care există un număr mare de agenți, acest concept poate fi foarte solicitant pentru sistemul pe care rulează aplicația și se recomandă folosirea câtor mai puține thread-uri. Soluția pentru rezolvarea acestei probleme este **actualizarea agenților prin parcurgerea**

fiecăruia pe același thread. De exemplu se pot pune toți agenți într-o listă, se parcurge lista și se incrementează poziția acestora la fiecare pas de simulare. Astfel, am rezolvat problema folosirii mai multor thread-uri rezumându-ne doar la unul singur. Se recomandă folosirea unui singur thread deoarece majoritatea librăriilor pentru afișarea grafică sau calcularea timpului folosesc în spate thread-uri. Un concept important legat de thread-uri este acela de sleep sau pauză. Fiecare limbaj de programare are diferite metode pentru această caracteristică. Este nevoie de acest concept deoarece un thread se poate „opri” atunci când este nevoie (mai exact atunci când în cod se ajunge la funcția respectivă pentru acest lucru), pentru a procesa alte task-uri [13]. În cazul unui simulator în care cu ajutorul unui thread se produce animația, se poate adăuga acest concept de sleep pentru procesare sau generare de rezultate în timp real. Bineînțeles că această pauză nu se observă cu ochiul uman deoarece se întâmplă foarte repede, dar este benefică pentru dezvoltarea unui simulator.

În capitolul ce urmează voi prezenta soluția propusă de mine pentru simularea unui trafic, procesarea datelor și afișarea rezultatelor.

Un simulator trebuie să facă o simulare cât mai corectă și să producă rezultate cât se poate de valide.

4.8. Suport tehnologic pentru aplicația de simulare

4.8.1. Java

Limbajul de programare **Java** este unul dintre cele mai folosite limbaje de programare bazate pe OOP (Oriented Object Programming). În plus este unul dintre cele mai ușoare limbaje de programare de învățat și folosit pentru dezvoltarea aplicațiilor deoarece a fost construit pe baza limbajului C (un limbaj care după părerea mea ar trebui să știe orice programator la început) adăugându-se în plus concepte de programare orientată pe obiect și funcționalități în plus.

Orice mediu hardware sau software în care rulează un program este cunoscut sub numele de **platformă**. Deoarece Java are un mediu de execuție (Java Runtime Environment), mediu de dezvoltare (Java Development Kit) și API (Application Programming Interface), se numește platformă. [14]

Există multe dispozitive în care Java este utilizat în prezent. Unele dintre ele sunt după cum urmează: Aplicații desktop, cum ar fi, media player, Antivirus, simulatoare etc.; Aplicații Web; Telefon; Embedded System; Robotica; Jocuri etc.

Așa cum prezintă autorii site-ului în [14], învățarea limbajului Java „is a MUST” pentru devenirea unui inginer software. Avantajele învățării acestui limbaj sunt următoarele: „orientat pe obiecte, independent de platforma, simplu, sigur, portabil și robust”. Aplicațiile dezvoltate în Java pot fi: „multithreaded, interpretate (de către un interpretor), distribuite și dinamice” [14].

Fiind un limbaj de programare bazat pe obiecte, este unul dintre cele mai potrivite limbaje pentru dezvoltarea unei aplicații de tip agent deoarece integrează toate principiile **programării orientate pe obiecte**.

4.8.2. Java Swing

Din punctul meu de vedere, un simulator trebuie să beneficieze și de o parte grafică unde se poate observa evoluția agenților în timp.

Acesta este un API ce oferă dezvoltatorului o suită de componente pentru a dezvolta o aplicație GUI (Graphical User Interface).

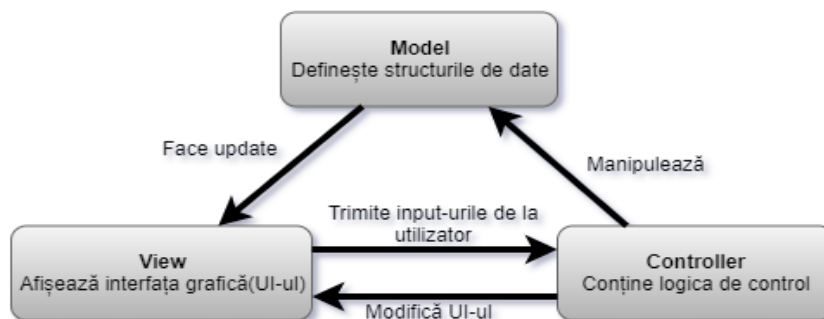
Am ales acest tool deoarece îl stăpânesc cel mai bine și l-am studiat în facultate. Tool-ul ofera butoane, panel-uri, text field-uri, combo box-uri și multe altele făcând interacțiunea cu programul cat mai ușoară și intuitivă. Un mare avantaj al acestui instrument este acela că se pot dezvolta aplicații GUI pentru orice sistem de operare, singura condiție este că acel sistem trebuie să aibă instalat Java Development Kit ce poate rula aplicații Java într-un mediu intern. [15]

Alte avantaje ale acestui tool sunt: **configurabil**(dezvoltatorul poate modifica design-ul componentelor implicate oferite de protocol), **interfață ușor de utilizat**(protocolul dispune componentele de bază pentru dezvoltarea unei aplicații grafice), **extensibil**(fiind un protocol ce se bazează pe componente, dezvoltatorul poate crea singur diferite componente folosind clasa JComponent și poate adăuga sau modifica funcționalitățile unui component), **specializat pentru dezvoltarea aplicațiilor de tipul Model-View**(model arhitectural ce face dezvoltarea aplicației mult mai ușoară și mai logică. Acest model este folosit și în proiectul dezvoltat de mine și este prezentat în Capitolul 5) [15].

Așadar, acest tool beneficiază de toate cerințele pentru dezvoltarea unui simulator ce are și o parte grafică.

4.8.3. Model-View-Controller(MVC)

Modelul arhitecturii MVC transformă dezvoltarea de aplicații complexe într-un proces mult mai ușor de gestionat. Permite mai multor dezvoltatori să lucreze simultan la aplicație [16]. După cum se poate observa în Figură 4.16, modelul este backend-ul care conține logica datelor, view este interfața grafică(UI-ul) și controller-ul este „creierul” aplicației și care controlează modul în care sunt afișate datele. Așadar, acest model arhitectural este cel mai potrivit pentru dezvoltarea unei aplicații desktop cu o interfață grafică.



Figură 4.16. Model-View-Cotroller

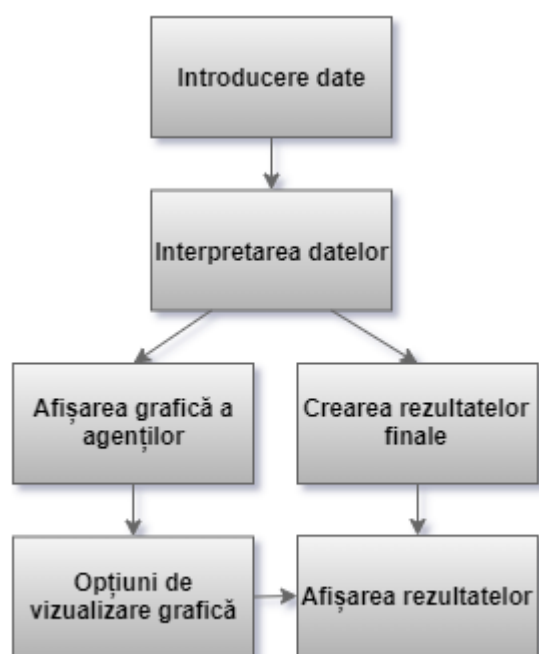
Capitolul 5. Proiectare de Detaliu și Implementare

Am dezvoltat o aplicație desktop în limbajul de programare Java ce poate fi rulată pe orice sistem de operare cu ajutorul mediului de programare(JDK) menționat în capitolul **Error! Reference source not found..** Am ales acest limbaj de programare deoarece este foarte ușor de dezvoltat aplicații orientate pe obiect, îl stăpânesc cel mai bine și majoritatea proiectelor din facultate le-am dezvoltat folosind acest limbaj.

Pentru a începe dezvoltarea unei astfel de aplicații, s-au luat în vedere caracteristicile discutate în capitolul 4.2. Mai jos sunt prezentate etapele dezvoltării acestei aplicații, de la faza de proiectare(schema generală a aplicației și arhitectura acesteia) până la rezultatul finit(clasele implementate și partea grafică).

5.1. Schema generală a aplicației

Pentru început, m-am gândit care sunt atribuțiile user-ului și cum fac ca aplicația să fie cât mai intuitivă și mai ușor de folosit. Mai mult de atât să producă rezultate logice, corecte și ușor de înțeles. În figura Figură 5.1 sunt prezentate situațiile de la care am pornit și pe care le-am luat în calcul la dezvoltarea aplicației. Dezvoltarea aplicației a fost făcută top-down, adică am stabilit funcționalitățile aplicației și le-am implementat în ordinea propusă. Nu am ales rezolvarea mai multor task-uri în paralel deoarece nu doream să se creeze confuzie atunci când combinam componentele dezvoltate (de exemplu într-o săptămână să lucrez la interpretarea datelor și afișarea grafică a agenților în paralel). Atunci când am ajuns la un anumit task, m-am focusat strict pe acel task și am încercat să-l implementez cât mai bine și mai logic.



Figură 5.1. Schema generală

dezvoltate în capitolul 3.2. Pentru a adapta aplicația la viața reală am ales următoarele tipuri de **mapări**:

Introducerea datelor se face printr-un fișier JSON. Am ales acest tip de fișier deoarece este unul dintre cele mai folosite și mai ușor de înțeles deoarece se folosește conceptul de cheie:valoare. Mai jos va fi explicat în detaliu care este structura fișierului JSON și care sunt cheile folosite pentru a înțelege aplicația cu tipurile de date care urmează să fie interpretate.

Interpretarea datelor se face la nivel de aplicație. În Java am declarat o clasă ce se ocupă strict cu citirea datelor din fișier și crearea structurilor de date ce ajută la implementarea algoritmilor, partea grafică și generarea rezultatelor.

După ce se citesc și se interpretează datele, acestea produc **partea grafică și creează rezultate finale** ce pot fi analizate. Am ales ca mediul în care se dezvoltă agenții să fie unul 2D și să fie asemănător cu cele

- Un metru din viața reală reprezintă un pixel în aplicație
- O secundă din viața reală reprezintă aproximativ(8000-9000 de milisecunde în aplicație. În aplicație apar diferite întârzieri din cauza rulării de aceea nu poate fi exact o secundă din viața reală)
- O mașină este reprezentată sub forma unui pătrat
- O stradă este reprezentată sub forma unor dreptunghiuri
- Semafoarele sunt reprezentate sub forma unor dreptunghiuri ce își schimbă culorile(roșu/verde) după un anumit timp
- Distanța unei străzi este reprezentat de lungimea(height) sau lățimea(width) acesteia
- Semafoarele apar doar la două tipuri de străzi(intersecție cu 3 străzi și intersecție cu 4 străzi)
- Mașinile se opresc una în spatele alteia sau la semafor la 5 pixeli(adică la 5 metri)
- Mașinile așteaptă la semafoare în funcție de timpul din simulator
- Rezultatele finale din simulator sunt mapate pentru lumea reală(timpul, distanța parcursă, viteza medie)
- Datele din simulator nu sunt vizibile în timpul simulării pentru a nu crea confuzie, sunt vizibile doar după ce se termină simularea și sunt mapate pentru un scenariu din lumea reală.

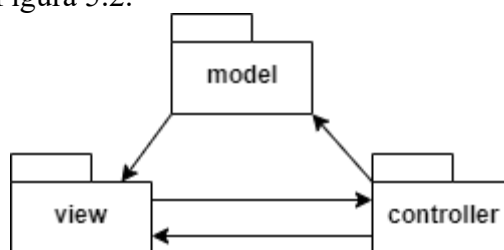
Toate aceste caracteristici sunt descrise în detaliu la partea de implementare cu exemple de cod. Așadar, acestea au fost primele decizii luate de mine înainte să scriu propriu-zis cod.

5.2. Proiectarea aplicației

Pentru dezvoltarea unei aplicații software, una dintre principalele caracteristici este cea a creării diagramelor arhitecturale. Acestea oferă o viziune mult mai clară a întregii aplicații și mai mult de atât, arată comunicarea dintre modulele acesteia. Acestea m-au ajutat să fac dezvoltarea aplicației mult mai repede, eficient și corect. Mai jos sunt prezentate următoarele tipuri de diagrame: **diagrama de pachete și diagrama de clase** specifice aplicației.

5.2.1. Diagrama de pachete

Fiind o aplicație de tip desktop una dintre alegerile importante de dezvoltare a fost alegerea **modelului arhitectural**. Modelul arhitectural ajută la alegerea alternativelor de design care fac un sistem reutilizabil și evitarea alternativelor care compromit reutilizarea. Mai simplu spus, modelul arhitectural ajută pe designer să obțină mai rapid un design „corect” [17]. Eu am ales Model View Controller deoarece majoritatea aplicațiilor desktop cu interfață grafică folosesc acest model și este cel mai logic de folosit pentru dezvoltarea unei aplicații de acest gen. Diagrama de pachete poate fi vizualizată în Figură 5.2.

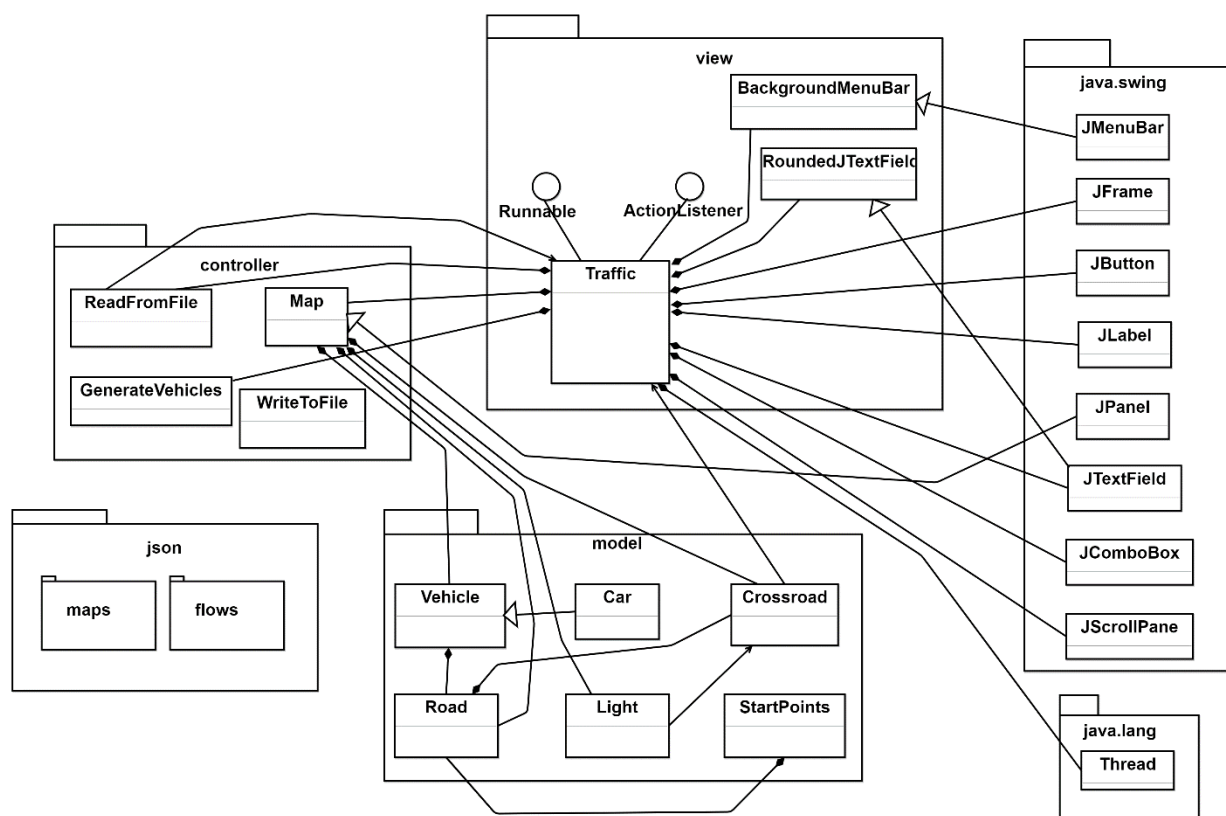


Figură 5.2. Diagrama de pachete

5.2.2. Diagrama de clase

Implementarea diagramei de clase este una dintre cele mai importante părți când vine vorba de proiectarea software a unei aplicații. Aceasta descrie clasele și relațiile dintre acestea din aplicație. Diagrama de clase arată atributele și metodele din fiecare clasă, și cum sunt declarate acestea (privat sau public) oferind o viziune de ansamblu a întregii aplicații. Diagrama de clase poate fi văzută în Figură 5.3. Deoarece în fiecare clasă sunt multe metode și atribute și nu se pot observa cum trebuie. Așa că am ales să prezint diagrama de clase fără numele acestora deoarece urmează să prezint în capitolul 5.3 cum este implementat fiecare modul. Mai jos sunt prezentate relațiile dintre clase și din ce pachet fac parte. Aplicația este formată din patru module principale:

- **Pachetul view:** în acest modul se află clasa principală Traffic unde este și funcția main (de unde începe aplicația). Clasa Traffic implementează interfețele Runnable (pentru a putea începe rularea unui thread) și ActionListener (ce face aplicația să fie dinamică și să „asculte” comenzile utilizatorului). După cum se poate observa, în clasa Traffic sunt declarate majoritatea celorlalte clase. Aceste declarații sunt descrise sub forma relației de compoziție (săgeata ce are la capăt un romb).
- **Pachetul controller:** în acest modul sunt declarate clase ce face posibilă simularea și cum evoluează agenții în timp. Mai mult de atât, sunt clasele ReadFromFile și WriteToFile ce procesează fișierele externe a aplicației. Clasa Map actualizează stările pentru fiecare agent din aplicație și este una dintre cele mai importante clase ale aplicației.
- **Pachetul model:** modul ce conține toți agenții din cadrul simulatorului. Aici se află clasele ce conțin informații despre fiecare model în parte (de exemplu coordonatele, distanțele, timpul, viteza).
- **Pachetul json:** conține fișierele JSON din care se citesc datele.



Figură 5.3. Diagrama de clase

5.3. Implementarea modulelor

În acest subcapitol se va prezenta în detaliu fiecare clasă în parte cu metodele și attributele specifice fiecăruia.

Fiind o aplicație de tip desktop, modelul arhitectural cel mai potrivit pentru o astfel de aplicație este Model View Controller. Astfel, am împărțit aplicația în trei pachete cu nume specific. Modelul general pentru acest model arhitectural a fost prezentat în 5.2.2. sub forma unei diagrame de clase. Deoarece diagrama de clase avea multe componente caracteristice, am ales să fac o diagramă de clase fără attributele și metodele respective fiecărei clase. Așadar, în acest capitol voi prezenta fiecare modul cu attributele și metodele acestora cu diagramele respective. În diagramele respective o metodă sau atribut privat începe cu simbolul „-” și o metodă sau atribut public începe cu simbolul „+”.

Formatul unui atribut este următorul: +/- numeAtribut : tipulAtributului.

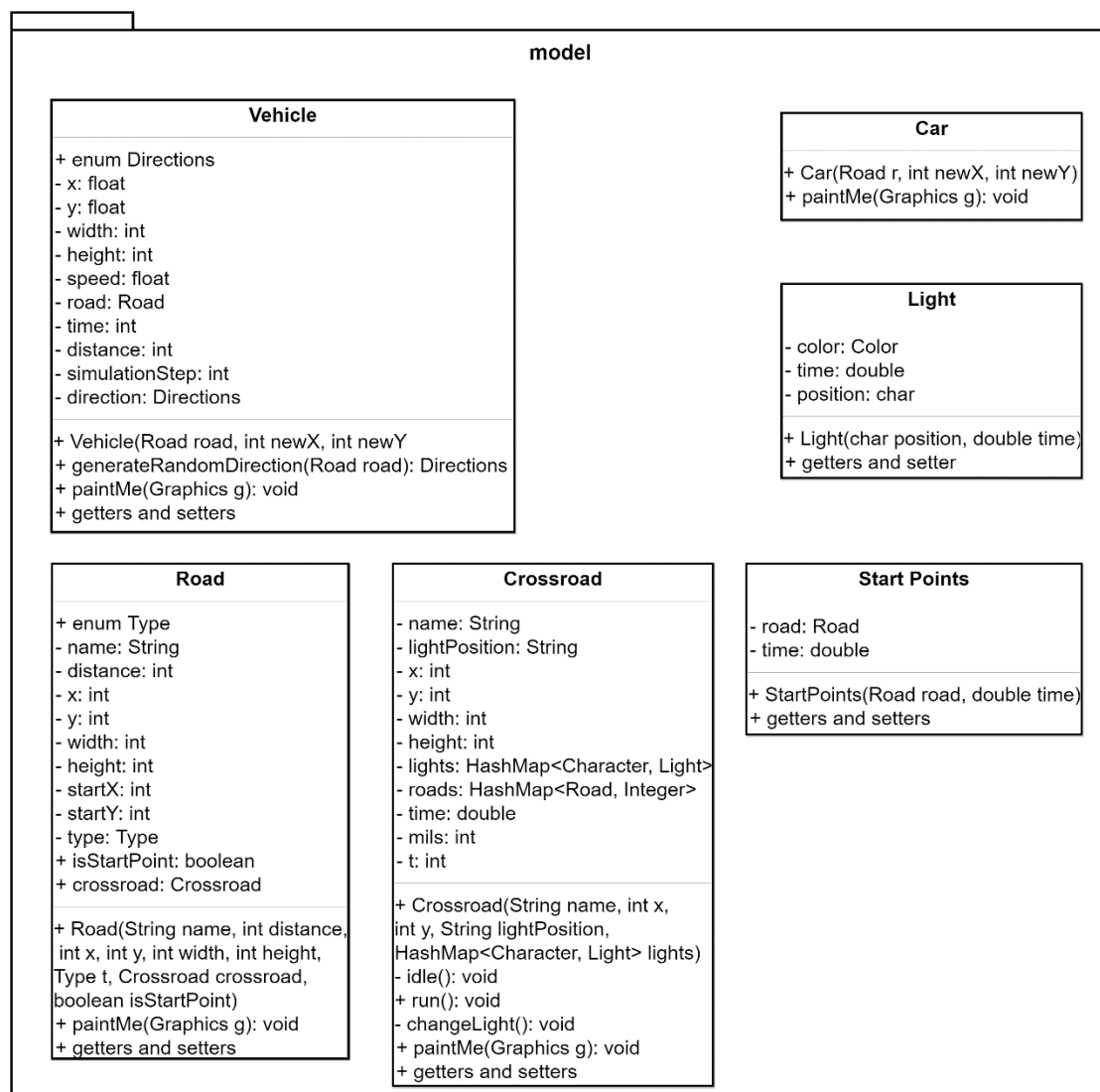
Formatul unei metode este următorul: +/- numeMetoda(attributeleMetodei) : ceReturneazăMetoda.

5.3.1. Model

Pachetul „model” reprezintă implementarea logică a fiecărui agent ce participă la simulare. Prin urmare, există șase clase ce fac parte din acest modul:

- Clasa Vehicle: reprezintă superclasa ce descrie fiecare vehicul din simulator. Se creează câte un obiect de acest tip la fiecare N secunde pe o anumită stradă în funcție de datele din fișierele din directorul „flow”, date ce sunt introduse de utilizator.
- Casa Car: este o subclasă a superclasei Vehicle. Aceasta reprezintă o mașina din simulator sub forma unui pătrat de culoare galbenă. În această clasă sunt declarate dimensiunile mașinii și strada de care aparține mașina respectivă.
- Clasa Road: reprezintă străzile ce formează o hartă în simulator. Acestea sunt reprezentate de dreptunghiuri și delimitate de linii albe. Mai mult de atât, fiecare drum are o anumită distanță pentru a calcula mai ușor distanțele totale ale mașinilor. Fiecare drum începe cu o intersecție sau nimic(caz în care se generează mașini pe acesta) și se termină cu o altă intersecție/nimic(caz în care este un drum final).
- Clasa Crossroad: reprezintă intersecțiile din hartă ce au atribuite semafoare. Pot fi de trei tipuri și se desenează în funcție de câte străzi intră în intersecție. Pot intra două, trei sau patru drumuri într-o intersecție. Atunci când intră două drumuri în intersecție, aplicația nu generează semafoare pentru drumurile respective.
- Clasa Light: reprezintă semafoarele dintr-o intersecție. Acestea au atribuite o anumită culoare și un anumit timp ce face culoarea să se schimbe.
- Clasa StartPoints: reprezintă străzi pe care se generează mașini. Punctele de pornire pentru mașini sunt străzi ce au ca startPoint = null în fișiere JSON. Adică drumul nu iese dintr-o intersecție anume.

Metodele și attributele specifice fiecărei clase de mai sus sunt prezentate în Figură 5.4.



Figură 5.4. Pachetul model

5.3.1.1. Clasa Vehicle

Această clasă descrie obiectele principale ce evoluează în timp în simulator și produce rezultate pe baza datelor de mediu. Mai exact, în această clasă se rețin datele ce se procesează și se produc rezultate pe baza acestora. Aceste date sunt prelucrate în timpul simulării în diferite clase.

Atributele ce se populează atunci când se declară un obiect(adică atributele din cadrul constructorului) de tipul Vehicle sunt următoarele:

- **road**: reprezintă drumul atribuit vehiculului respectiv. Cu alte cuvinte, o mașină se află pe un anumit drum.
- **x** și **y**: reprezintă coordonatele formei geometrice ce reprezintă un vehicul.

Atributele importante declarate în cadrul clasei sunt următoarele:

- **enum Directions**: reprezintă o enumerație de direcții ce este declarată în clasa Vehicle. Aceasta putea fi declarată ca și o clasă separată. Direcțiile ce fac parte din această enumerație sunt: FORWARD, LEFT,

RIGHT, END, unde END este atribuit atunci când strada pe care se află mașina este end point.

- **width** și **height**: reprezintă lungimea și lățimea formei geometrice
- **speed**: reprezintă viteza agentului. Mai exact, în implementare, reprezintă numărul de pixeli ce se incrementează la fiecare pas de simulare. Așadar, cu ajutorul acestei valori, se produce mișcarea fiecărui agent din simulare
- **time**: reprezintă timpul total pentru fiecare vehicul, adică la fiecare pas de simulare, se actualizează timpul
- **distance** și **simulationTime**: reprezintă distanțele totale fiecărui vehicul. Diferența dintre distance și simulationTime este aceea că distanța este o variabilă ce se actualizează cu distanțele drumurilor respective și simulationTime este variabila ce se incrementează la fiecare pas de simulare
- **direction**: reprezintă direcția actuală a mașinii

Metoda importantă din cadrul acestei clase este:

- **generateRandomDirections(Road road)**: reprezintă metoda ce generează următoarea direcție a mașinii în funcție de ce direcții sunt în următoarea intersecție. Acest lucru se realizează după o secvență de if-uri, după care la finalul acestor secvențe se execută următoarele linii de cod ce generează **random** următoarea direcție. Aceasta direcție se ia din lista de direcții aux:

```
1. int rnd = new Random().nextInt(aux.size());
2. return aux.get(rnd);
3.
```

5.3.1.2. Clasa Car

Această clasă este o subclasă a clasei Vehicle. Am ales să implementez clasa Car deoarece este mai logic să fie subclase ce reprezintă mai multe tipuri de vehicule.

Atunci când se creează un obiect de tipul Car, se populează sau se actualizează variabilele din constructorul clasei părinte, adică variabilele road, x și y. Acest lucru este posibil cu ajutorul metodei din limbajul Java **super**. Atunci când se declară obiectul se apelează metoda super(road, x, y) din constructor.

Pe lângă valorile road, x și y, se specifică lungimea și lățimea vehiculului (adică cum să arate forma geometrică din simulator) și viteza acesteia. Așadar, un agent de tipul Car va avea următoarele valori:

- width = 5;
- height = 5;
- speed = 1.4;

Prin urmare, un obiect va fi reprezentat de un pătrat de 5 pixeli ce are o viteză inițială de 1.4 (adică la fiecare pas de simulare se incrementează 1.4 pixeli)

5.3.1.3. Clasa Road

Cu ajutorul acestei clase se poate crea diferite hărți în funcție de datele introduse de utilizator. Atunci când utilizatorul dorește să creeze o hartă așa cum dorește, acesta formează o listă de străzi ce sunt interconectate cu ajutorul intersecțiilor.

Atributele din cadrul constructorului sunt următoarele:

- **name:** reprezintă un string cu numele străzii. Aceasta variabilă ajută la identificarea străzii în timpul simulării și face crearea unei hărți mult mai ușor de realizat
- **distance:** reprezintă distanța drumului respectiv. Această valoare se citește din fișierul JSON dacă strada respectivă este start point sau end point. În cazul în care strada se află între două intersecții, valoarea se calculează automat.
- **x și y:** reprezintă coordonatele formei geometrice ce reprezintă o stradă.
- **width și height:** reprezintă lungimea și lățimea formei geometrice
- **type:** reprezintă tipul străzii respective. Aceasta este o valoare din enumerația Type
- **crossroad:** reprezintă intersecția atribuită străzii.
- **isStartPoint:** reprezintă un boolean ce specifică dacă strada respectivă este start point sau nu, adică dacă să se genereze mașini pe această stradă sau nu.

Alte attribute din cadrul clasei Road ce descriu clasa sunt:

- **enum Type:** este o enumerație de tipuri de străzi ce descriu în ce sens circulă mașinile pe strada respectivă. Elementele din enumerație sunt: RL(mașinile circula de la dreapta la stânga), LR(mașinile circulă de la stânga la dreapta), UD(mașinile circulă de sus în jos), DU(mașinile circulă de jos în sus). Această enumerație putea fi declarată ca și o clasă separată.
- **startX și startY:** reprezintă coordonatele din care se generează un vehicul în cazul în care strada este start point sau end point. Am ales să adaug aceste două valori deoarece vehiculele se generează din capetele dreptunghiului
- **isEndPoint:** reprezintă un boolean ce specifică dacă strada respectivă este un end point sau nu. Dacă o stradă este end point, mașinile nu au o direcție următoare și după ce termină de traversat strada respectivă, acestea sunt șterse din lista de vehicule din simulator și generează rezultatele necesare.

Această clasă este formată din metode de get și set pentru variabilele descrise mai sus.

5.3.1.4. Clasa Crossroad

Această clasă descrie o intersecție din harta generată. O intersecție este reprezentată sub forma unui pătrat de culoare gri și la drumurile ce intră în intersecție este atribuit câte un semafor.

Atributele din cadrul constructorului sunt următoarele:

- **name:** reprezintă un string cu numele intersecției. Am ales să pun câte un nume la fiecare intersecție pentru a face desenarea unei hărți mai ușor din fișierul JSON. Când se scriu valorile pentru generarea hărții, se poate vedea mult mai ușor ce drumuri intră și ies dintr-o intersecție ce are un nume respectiv.
- **x și y:** reprezintă coordonatele formei geometrice ce reprezintă o stradă.
- **lightPosition:** reprezintă un string ce poate avea următoarele caractere: „U”, „D”, „R”, „L” necontând ordinea. Aplicația ia caracterele din acest string și generează câte un semafor pe poziția reprezentată de caractere.

- **lights**: reprezintă un hashMap în care cheia este reprezentată de un caracter (poziția semaforului ce este un caracter din string-ul lightPosition) și valoarea este reprezentată de un obiect de tipul Light

Atributele importante declarate în cadrul clasei Crossroad sunt următoarele:

- **width** și **height**: reprezintă lungimea și lățimea formei geometrice. Acesta este reprezentată de un pătrat cu lungimea de 30 de pixeli.
- **roads**: reprezintă un hashMap în care cheia este reprezentată de un obiect de tipul Road și valoarea este reprezentată de un întreg ce poate avea două valori: 0 în cazul în care drumul respectiv intră în intersecție și 1 în cazul în care drumul respectiv iese din intersecție.
- **time**: reprezintă o secundă. Aceasta valoare ajută la modificarea semafoarelor la N secunde
- **t**: este un contor ce face schimbarea semafoarelor în sensul acelor de ceasornic. De exemplu când $t = 0$ semaforul din partea de sus este verde, când $t = 1$ semaforul din partea dreapta este verde și așa mai departe
- **mils**: reprezintă numărul de milisecunde. Acesta ajută la calcularea secundelor cu ajutorul funcției run() descrisă mai jos.

Metodele importante din cadrul acestei clase sunt:

- **idle()**: este o metodă în care se parcurge hashMap-ul lights și se setează la toate semafoarele culoarea roșie. Această metodă este apelată în constructor de fiecare dată când se începe simularea.
- **run()**: este o metodă ce verifică la fiecare pas de simulare dacă variabila mils a ajuns la 100 (adică la o secundă). Când s-a ajuns la un anumit număr de secunde, semafoarele se schimbă. Implementarea acestei metode este următoarea:

```

1. public void run() {
2.     if (mils == time) {
3.         changeLight();
4.         mils = 0;
5.     } else {
6.         mils++;
7.     }
8. }
```

- **changeLight()**: este metoda ce comută semafoarele în funcție de variabila t. După o secvență de if-uri ce verifică direcțiile din string-ul lightPosition, se atribuie valori la fiecare semafor și se incrementează variabila t.

5.3.1.5. Clasa Light

Este clasa ce reprezintă un singur semafor din hartă. Am integrat această clasă deoarece este mai ușor de memorat valorile semafoarelor într-o clasă separată și să fie accesate cu ajutorul metodelor de get.

Atributele din cadrul constructorului sunt următoarele:

- **position**: reprezintă un caracter din string-ul prezentat mai sus(lightPosition) ce memorează poziția semaforului într-o intersecție.
- **time**: reprezintă numărul de secunde pentru a modifica culoarea semaforului

Un alt atribut important al clase `Light` este **color** ce reprezintă culoarea semaforului. Acesta poate avea două culori: roșu și verde. Aceasta variabilă reprezintă un obiect al clasei `Color` și se poate atribui o culoare foarte ușor folosind variabilele statice din clasa `color`. Atribuirea unei culori se face în următorul mod:

```
1. color = Color.green;
```

5.3.1.6. Clasa *StartPoints*

Am ales să implementez o clasă ce reține strazile pe care se generează mașini și sunt puncte de start. Când se citește din fișier, pot fi mai multe străzi pe care se generează mașini. Așadar, am creat această clasă pentru a crea o listă de puncte de start și cu ajutorul căruia se generează mașini pe strazile respective.

Atributele din cadrul constructorului sunt următoarele:

- **road**: ce reprezintă strada ce este punct de start
- **time**: reprezintă timpul (în secunde) în care se generează mașini pe strada respectivă. Această valoare se citește din fișierele JSON din directorul `flows`

Această clasă nu are alte atribute în afară de cele din constructor.

În fiecare metodă din pachetul `view` am implementat câte o metodă ce face desenarea fiecăruia în interfața grafică. Aceasta metodă se numește **paintMe(Graphics g)**. În această metodă se pot adăuga diferite componente pentru variabila `g` cum ar fi (componentele se adaugă în funcție de ordinea lor în cod):

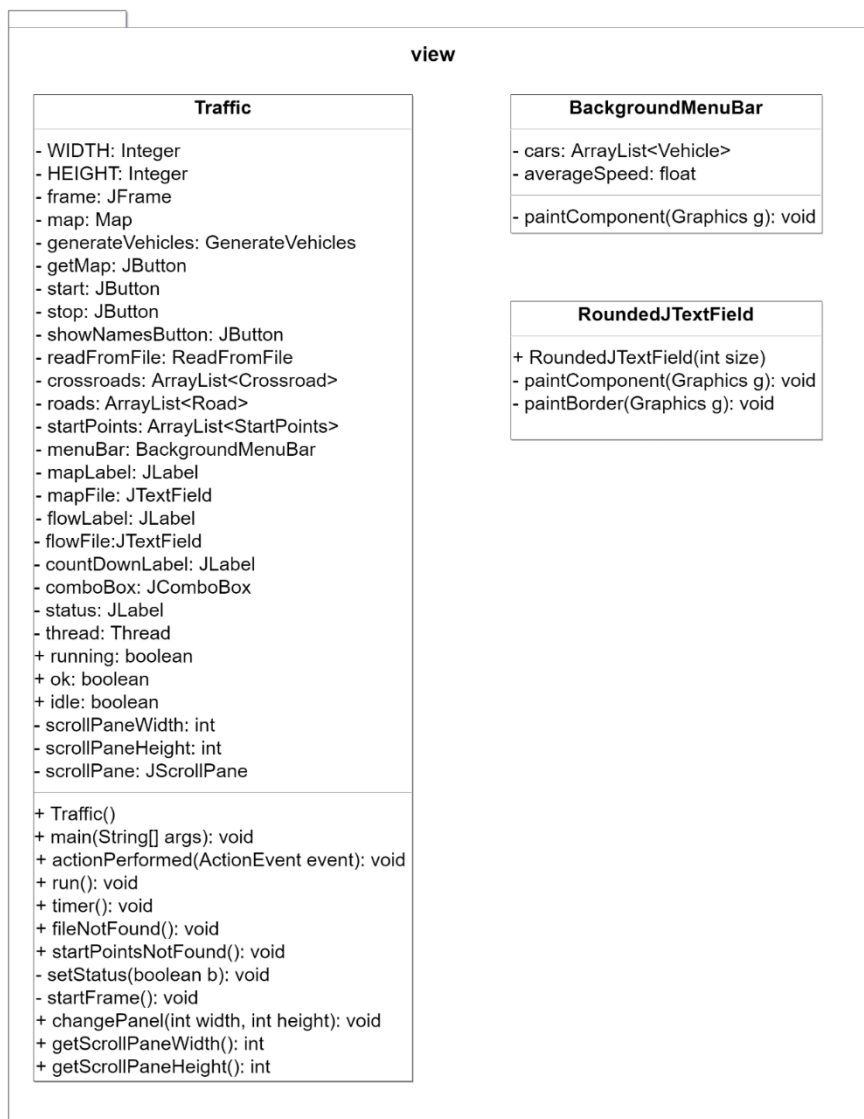
- formele geometrice: cu ajutorul metodei **g.fillRect((int) x, (int) y, width, height);**
- se atribuie culori pentru componente: **g.setColor(Color.red);**
- se scrie text la o anumită poziție: **g.drawString("F", (int) x, (int) y);**
- se atribuie un anumit font pentru text: **g.setFont(new Font("Helvetica", Font.PLAIN, 9));**

5.3.2. View

Pachetul view reprezintă partea în care se implementează interfața grafică a aplicației. În această clasă se folosesc metode din biblioteca Swing prezentată în capitolele precedente, fără această bibliotecă nefiind posibilă dezvoltarea unei aplicații cu user interface. În acest pachet sunt declarate formele geometrice, butoanele, text field-urile, label-urile, etc. Acest pachet este reprezentat în Figură 5.5 și Figură 5.4.

Prin urmare, există trei clase ce fac parte din acest modul:

- Clasa Traffic: este clasa în care se declară componentele din librăria Swing și se setează mărimile și caracteristicile principale ale frame-ului în care se desfășoară simularea(de exemplu care sunt dimensiunile frame-ului, să se seteze ca acesta să nu-și poată modifica dimensiunile, etc.)
- Casa BackgroundMenuBar: această clasă crează meniul principal al aplicației în care se adaugă diferite componente(butoane, text field-uri, etc). Această extinde clasă JMenuBar.
- Clasă RoundedJTextField: este clasa ce face text-fieldul să nu aibă colțuri drepte. Am implementat o astfel de clasă pentru estetica.



Figură 5.5. Pachetul view

5.3.2.1. Clasa Traffic

În această clasă se declară componentele din pachetul Java Swing. Toate clasele din Java Swing sunt subclase ale clasei Component, adică toate se pot personaliza așa cum dorește dezvoltatorul. Tot în această clasă se află și metoda **main**, prima metodă ce se apelează atunci când se rulează aplicația.

Eu am ales următoarele componente cu care utilizatorul să poată interacționa:

- **frame**: această componentă reprezintă fereastra ce se deschide o dată ce se rulează aplicația. În aceasta se introduc componentele ce urmează să fie folosite de către utilizator. Caracteristicile pentru un frame Java Swing sunt: se setează o dimensiune pentru acesta (înălțime și lățime), se setează un layout (acesta reprezintă cum sunt aranjate componentele în frame), se setează ca dimensiunile acestuia să nu poată fi modificate cu mouse-ul (setResizable(false)), atunci când se închide frame-ul (când se apasă pe butonul „X”) să se oprească și rularea aplicației, acesta trebuie să fie vizibil pentru utilizator (prin metoda setVisible(true)), și să se adauge diferite componente în acesta cu ajutorul metodei add(...). prin metoda „add” se pot adăuga butoane, text field-uri, layout-ul dorit, etc. Când se declară această componentă, trebuie scrise următoarele caracteristici:

```
1. frame.setSize(WIDTH, HEIGHT);
2. frame.setLayout(new BorderLayout());
3. frame.setResizable(false);
4. frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
5. frame.setVisible(true);
6. frame.add(<componenta>);
```

- **butoane**: în aplicație am implementat butonul de generare hartă (când se apasă, se citește din fișierele JSON și se generează harta), butonul de start (o dată apăsat, începe simularea), butonul de stop (oprește simularea) și butonul de afișare sau ascundere nume din simulator (acesta se comportă ca un toggle pentru aplicație),
- **label-uri**: acestea informează utilizatorul ce să introducă în câmpurile text din aplicație. Acestea reprezintă un text informativ în aplicație ce nu poate fi selectat sau modificat de utilizator. Am implementat un label ce are ca simbol „✓” și care se face verde atunci când utilizatorul a introdus numele fișierelor corect și poate apăsa pe butonul de start pentru a putea începe simularea.
- **text field-uri**: sunt două la număr și se introduc în acestea numele fișierelor JSON din care să se citească datele, mai exact fișierul pentru harta și fișierul ce generează fluxul de mașini.
- **combo box**: acesta reprezintă un buton ce afișează mai multe variante pe care utilizatorul le poate alege. Aceste variante sunt viteza de simulare ce pot fi: „Slow”, „Normal”, „Fast”, „Super Fast” și „Get Results”. Pentru „a ști” aplicația ce selectează utilizatorul, am atribuit combo box-ului un listener ce reține alegerea făcută de utilizator. O dată ce utilizatorul a selectat viteza dorită din combo box, viteza mașinilor se modifică atunci când simularea este pornită sau nu prin atribuirea la variabilă simulationType diferite valori. Aceste valori reprezintă timpul în care thread-ul apelează metoda sleep.

```

1.  comboBox.addActionListener(e -> {
2.      if (comboBox.getSelectedIndex() == 0)
3.          simulationType = 25;
4.      if (comboBox.getSelectedIndex() == 1)
5.          simulationType = 10;
6.      if (comboBox.getSelectedIndex() == 2)
7.          simulationType = 5;
8.      if (comboBox.getSelectedIndex() == 3)
9.          simulationType = 1;
10.     if (comboBox.getSelectedIndex() == 4)
11.         simulationType = 0;
12. });

```

- **scroll pane:** acesta se declară numai în cazul în care utilizatorul introduce date ce depășesc mărimea inițială a panel-ului. Când acest lucru se întâmplă, fereastră poate fi navigată cu ajutorul barelor de scroll ce apar pentru a putea fi vizualizat comportamentul tuturor agenților din simulare, indiferent ce mărime au strazile declarate în fișierul JSON.

Pe lângă aceste variabile ce descriu componente din interfața grafică, sunt și alte variabile ce se ocupă de partea logică a aplicației:

- **crossroads, roads, startPoints:** sunt liste ce rețin ageții din simulare. Aceste liste sunt populate cu valorile citite din fișierul JSON și se trimit mai departe clasei Map ce se ocupă cu actualizarea fiecărui obiect din aceste liste.
- **thread:** este o instanță a clasei Thread și reprezintă thread-ul principal al aplicației. Metoda run() a clasei thread este suprascrisă cu comportamentul aplicației și acest thread face sleep la fiecare 10 milisecunde.
- **running:** reprezintă o variabilă booleană ce reține dacă simularea are loc sau aceasta este pusă pe pauză. Aceasta variabilă se schimbă în funcție de butoanele „Start” și „Stop”.
- **ok:** este tot o variabilă booleană ce reține starea aplicației. Mai exact, dacă nu se introduce numele fișierului corect, la apăsarea butonului de start nu va începe simularea.
- **map și generateVehicles:** aplicația având un model arhitectural Model-View-Controller, după cum se poate observa din diagrama de pachete, view-ul trebuie să comunice cu controller-ul. Așa că am declarat aceste două clase din controller deoarece controller-ul modifică diferite date din view și view-ul trebuie să actualizeze vizual aceste date. Clasa Map actualizează fiecare agent din simulator și produce mișcare și clasa GenerateVehicles generează vehiculele la o anumită perioadă de timp pe o anumită stradă.
- **simulationType:** reprezintă un întreg ce schimbă viteza simulării. Mai exact, această variabilă se modifică atunci când utilizatorul selectează din combo box viteza dorită. Valoarea reprezintă timpul (în milisecunde) în care thread-ul apelează metoda run(). Aceasta poate avea cinci valori: 25 (animația este mai încetă deoarece funcția run este apelată mai greu), 10 (este viteza normală pentru simulare), 5, 1 (animația este mult mai rapidă deoarece metoda se apelează la 5 sau 1 milisecunde) și 0 în care simularea are loc instant și se produc rezultatele pe loc. Am inclus această opțiune deoarece am luat cazul în care utilizatorul nu are răbdare să urmărească simularea și să aibă rezultatele instant.

În această clasă sunt declarate metode ce produc mișcarea obiectelor din simulator. Metodele principale din clasa Traffic sunt:

- **main(String[] args):** este prima metodă ce se apelează când se deschide aplicația. În această metodă se declară un obiect de tipul Traffic pentru a deschide interfața grafică pentru utilizator
- **actionPerformed(ActionEvent event):** este metoda ce se apelează când se interacționează cu o componentă ce are atribuit un listener (face o acțiune atunci când utilizatorul folosește componenta respectivă, adică apăsă pe un buton, sau face click pe ecran). Implementarea acestei metode poate fi văzută la Anexa 1. În această metodă sunt declarate mai multe if-uri ce verifică ce componentă a folosit utilizatorul. De exemplu, când utilizatorul apasă pe butonul „Generate Map” se crează un obiect al clasei ReadFromFile, se apelează constructorul clasei ce face citirea din fișier și se populează listele declarate în clasa Traffic cu obiecte ce reprezintă agenții simulării. Un alt exemplu este atunci când utilizatorul apasă pe butonul de start, pornește thread-ul ce produce animația și ce face sleep la fiecare 10 milisecunde. Secvența de cod caracteristică acestei acțiuni este următoarea:

```
1.  if (event.getSource().equals(start)) {
2.      if (!running && ok) {
3.          running = true;
4.          thread = new Thread(this);
5.          thread.start();
6.      }
7.  }
```

Un alt exemplu este cel de afișare/ascundere a numelor pentru fiecare agent ce este următoarea:

```
1.  if (event.getSource().equals(showNamesButton)) {
2.      if (!showNames) {
3.          showNamesButton.setText("Hide Names");
4.          showNames = true;
5.          frame.repaint();
6.      } else {
7.          showNamesButton.setText("Show Names");
8.          showNames = false;
9.          frame.repaint();
10.     }
```

- **run():** este metoda ce se apelează la fiecare 10 milisecunde și este specifică clasei Thread. De aceea se suprascrie această metodă cu acțiunile ce vreau să le facă aplicația. (înainte de declararea metodei se pune cuvântul @Override). La fiecare apelare a acestei metode se execută secvențele de cod din acestea, mai exact se actualizează starea fiecărui agent, se incrementează contorul de timp și se generează fluxul de mașini pe străzile ce sunt start points. Implementarea acestei metode poate fi vizualizată în Anexa 1.
- **startFrame():** această metodă se apelează atunci când utilizatorul deschide aplicația. Este ca o pagină de pornire unde se poate vizualiza numele aplicației și o intersecție.
- **timer():** este metoda ce decrementează secunde afișate în aplicație. La fiecare 100 de milisecunde se decrementează variabilă countdown.

Acest lucru se face prin aflarea restului împărțirii valorii mils la 100. Atunci când restul împărțirii este 0, înseamnă că au trecut 1000 de milisecunde, adică o secundă. Am împărțit la 100 și nu la 1000 deoarece pasul de simulare este de 10 milisecunde și la 10 milisecunde se incrementează variabila mils.

- **fileNotFound()** și **startPointNotFound()**: sunt metode ce se apelează atunci când utilizatorul nu introduce numele fișierelor în câmpurile de text și informează utilizatorul că nu a introdus datele corespunzătoare. Exemplu de cod ce realizează informarea utilizatorului este:

```
1. JOptionPane.showMessageDialog(null,
2.     "Start points not found!\n" +
3.     "(Check file name OR road names from flow file)",
4.     "Error", JOptionPane.ERROR_MESSAGE);
```

- **changePanel(int width, int height)**: această metodă se apelează atunci când datele introduse de utilizator în fișierul JSON generează o hartă mai mare decât limitele implicite ale ferestrei în care se generează harta. Așadar, această funcție mărește panel-ul în funcție de cât de mult depășește dimensiunile ferestrei prin cu ajutorul metodei:

```
1. map.setPreferredSize(new Dimension(scrollPaneWidth, scrollPaneHeight));
```

- **setStatus(boolean b)**: această metodă schimbă culoarea simbolului „√” în verde când aplicația este pregătită de rulare.

5.3.2.2. Clasa BackgroundMenuBar

Aceasta este o clasă internă a clasei Traffic și extinde clasa JMenuBar. Am ales să fac o clasă proprie deoarece varianta implicită nu se potrivea interfața aplicației dezvoltată. Așadar în această clasă am declarat un atribut bgColor = Color.white ce reprezintă culoarea de fundal a barei de meniu.

Pentru a putea face modificări vizuale asupra barei de meniu, am suprascris metoda paintComponent după cum urmează:

```
1. @Override
2. protected void paintComponent(Graphics g) {
3.     super.paintComponent(g);
4.     Graphics2D g2d = (Graphics2D) g;
5.     g2d.setColor(bgColor);
6.     g2d.fillRect(0, 0, getWidth() - 1, getHeight() - 1);
7. }
```

5.3.2.3. Clasa RoundedJTextField

Această clasă face colturile câmpurilor de text rotunjite. De aceea, această clasă extinde clasa JTextField în care suprascrie metoda paintComponent(Graphics g) la fel ca în cazul de mai sus, numai că în loc de metoda de fillRect() am scris metoda: **g.fillRoundRect(0, 0, getWidth() - 1, getHeight() - 1, 5, 5);**

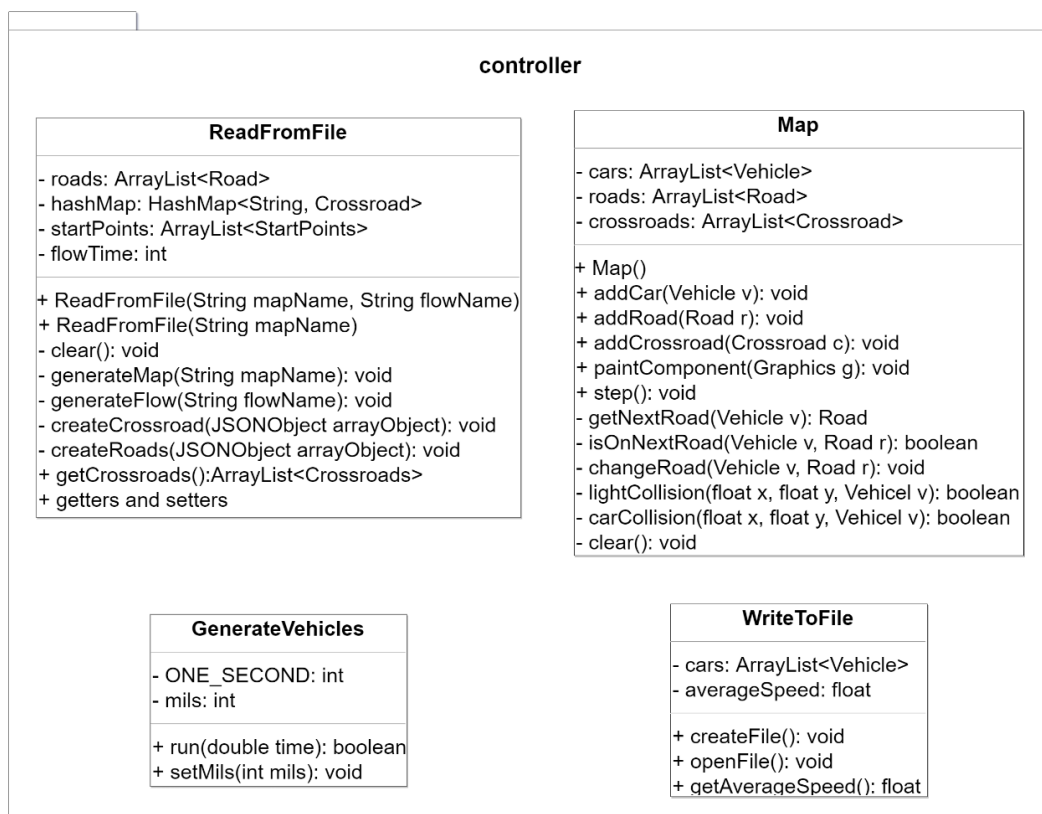
Pe lângă metoda paintComponent(Graphics g) am mai suprascris metoda paintBorder în care am modificat și marginile text field-urilor, prin micșorarea acestora cu un pixel. Am ales să fac acest lucru pentru estetică și pentru că marginile erau prea groase pentru meniul declarat mai sus.

5.3.3. Controller

Clasa controller reprezintă „creierul” aplicației. În această clasă se întâmplă partea logică a aplicației și descrie cum se întâmplă acțiunile din simulator. Acest modul este al treilea element din arhitectura Model-View-Controller. După cum îi spune și numele, acest modul controlează agenții/modelele din cadrul aplicației și acțiunile ce urmează să le facă acești agenți. Tot acest modul, după ce actualizează sau modifică datele, face update la view pentru a putea fi văzute grafic modificările făcute. În cadrul aplicației, în controller se incrementează poziția mașinilor, timpul semafoarelor și timpul de simulare, după care transmite aceste date la view și acesta le afișează în interfața grafică. Pachetul controller poate fi vizualizat în Figură 5.6.

Prin urmare, există patru clase ce fac parte din acest modul:

- Clasa Map: este cea mai importantă clasă a pachetului controller deoarece în această clasă se actualizează fiecare agent al simulării. Această clasă ia fiecare obiect din simulare și la fiecare pas de simulare îl actualizează la noua stare formând procesul de simulare.
- Clasa ReadFromFile: această clasă face posibilă citirea din fișierele JSON introduse de utilizator. Numele fișierelor sunt trimise ca argumente ale constructorului când se crează obiectul acestei clase. Această clasă folosește librării speciale ce face interpretarea datelor mai ușor de folosit și de interpretat. Aceste librării sunt: org.json.*;(ce prelucerează și formează obiecte în Java de tipul JSON) și java.io.*;(ce face posibilă citirea datelor din fișier)
- Casa WriteToFile: această clasă are metode ce formează fisietul text Results.txt și populează acest fișier cu rezultatele așteptate.
- Clasa GenerateVehicles: este clasa ce generează vehicule la N secunde.



Figură 5.6. Pachetul controller

5.3.3.1. Clasa Map

În această clasă se actualizează obiectele din pachetul model. La fiecare pas de simulare, se verifică și se actualizează starea agenților. Această clasă extinde clasă JPanel deoarece simularea are loc într-un panel. Astfel, în această clasă se introduc în panel următorii agenți: Car, Road și Crossroad.

Atributele din această clasă sunt:

- **cars:** reprezintă o listă în care se stochează obiectele de tip Car
- **roads:** reprezintă o listă în care se stochează obiectele de tip Road
- **crossroads:** reprezintă o listă în care se stochează obiectele de tip Crossroad

Metodele sunt următoarele:

- **metode ce adaugă agenții în interfața grafică, mai exact în panel:** în aceste metode se parcurg listele declarate mai sus și se adaugă în panel fiecare agent în parte.
- **step():** în această metodă se parcurge lista cars și pentru fiecare mașină, se incrementează poziția x sau y cu viteza specifică a acesteia. Această incrementare se face în funcție de tipul de drum pe care se află aceasta. Tot în această metoda se verifică dacă o mașină se află în intersecție se apelează metoda ce schimbă strada atribuită mașinii. Secvența de cod ce realizează acest lucru este următoarea:

```

1. // se afla in intersecție
2. if (v.getY() < v.getRoad().getCrossroad().getY() + 30) {
3.     Road r = getNextRoad(v);
4.     if (isOnNextRoad(v, r)) {
5.         changeRoad(v, r);
6.     } else {
7.         v.setY(v.getY() - v.getSpeed());
8.     }
9. } else {
10.    v.setY(v.getY() - v.getSpeed());
11. }

```

În această metodă se verifică pentru fiecare tip de stradă: „DU”, „UD”, „LR”, „RL”.

- **getNextRoad(Vehicle v):** este una dintre cele mai consistente metode din program deoarece în funcție de tipul străzii unei mașini, se parcurge lista de intersecții pentru a găsi drumul ce intră în intersecția respectivă și pentru a găsi drumul ce iese din intersecție în funcție de următoarea direcție a mașinii. Așadar, se verifică prima dată tipul străzii, după care se verifică care este următoarea direcție a mașinii. O dată ce s-a găsit if-ul respectiv, se parcurge lista de intersecții și în funcție de direcția mașinii se găsește drumul ce urmează să fie schimbat pentru mașina respectivă.
- **isOnNextRoad(Vehicle v, Road r):** reprezintă metoda ce poziționează mașinile pe următorul drum cum trebuie în funcție de poziția acesteia. Așadar în această metodă sunt patru if-uri (pentru fiecare tip de drum), și verifică dacă mașina se află la o anumită distanță față de poziția drumului pentru ca atunci când se schimbă strada pentru o mașină, aceasta să fie poziționată cum trebuie pe strada respectivă. If-urile ce verifică dacă atunci când o mașină se află la un anumit număr de pixeli față de o stradă sunt următoarele:

```

1.  if (r.getType() == Road.Type.LR) {
2.      return (int) v.getY() == r.getY() + 4 || (int) v.getY() == r.getY() + 5;
3.  }
4.  if (r.getType() == Road.Type.RL) {
5.      return (int) v.getY() == r.getY() + 5 || (int) v.getY() == r.getY() + 6;
6.  }
7.  if (r.getType() == Road.Type.UD) {
8.      return (int) v.getX() == r.getX() + 5 || (int) v.getX() == r.getX() + 6;
9.  }
10. if (r.getType() == Road.Type.DU) {
11.     return (int) v.getX() == r.getX() + 5 || (int) v.getX() == r.getX() + 6;
12. }
13.

```

- **changeRoad(Vehicle v, Road r):** este metoda ce modifică strada(ce este argument al obiectului Car) cu următoarea stradă ce se trimite ca argument. Implementarea acestei clase este scrisă în Anexa 1.
- **lightCollision(float x, float y, Vehicle v):** reprezintă metoda ce verifică dacă în fața unei mașini este un semafor. Aceasta metodă se apelează la fiecare pas de simulare pentru a „ști” mereu mașina ce se află în fața acesteia. Așadar sunt patru if-uri în funcție de tipul drumului ce verifică dacă poziția inițială a mașinii se află la un anumit număr de pixeli față de semaforul de pe strada respectivă. Dacă mașina este la numărul de pixeli respectiv și semaforul este roșu, mașina se oprește. În caz contrar, mașina intră în intersecție și se modifică strada atribuită acesteia.
- **carCollision(float x, float y, Vehicle v):** la fel ca metoda precedentă, la fiecare pas de simulare se verifică dacă se află o mașină în fața acesteia. Tot în această metodă se verifică dacă se produce blocaj, adică dacă o mașină „a intrat” în altă mașină. Atunci când se întâmplă acest lucru, simularea se blochează și utilizatorul este informat că s-a produs un blocaj pe o anumită stradă. Implementarea acestei clase este scrisă în Anexa 1.
- **clear():** metodă ce șterge datele din atributele clasei scrise mai sus. Am implementat acest lucru deoarece de fiecare dată când se face o nouă simulare, trebuie șterse datele precedente pentru a nu produce erori.

5.3.3.2. Clasa ReadFromFile

Această clasă este foarte importantă deoarece face posibilă citirea din fișier. În această clasă se folosesc pachetele org.json.*; pentru a face posibilă interpretarea datelor din fișier ca fiind unul JSON și să poată fi manipulate mai ușor și java.io.*; cu ajutorul căreia se face posibilă deschiderea și citirea fișierelor de hartă și de flux de mașini.

Atributele din cadrul constructorului sunt următoarele:

- **mapName:** este un șir de caractere ce reprezintă numele fișierului ce generează harta
- **flowName:** este un șir de caractere ce reprezintă numele fișierului ce generează fluxul de mașini

Atributele importante declarate în cadrul clasei Crossroad sunt următoarele:

- **roads:** reprezintă o lista în care se pun toate strazile ce se citesc din fișier. Acest argument este accesat din clasa Traffic
- **hashMap:** reprezintă un HashMap în care se stochează numele intersecțiilor sub forma de cheie și clasa intersecției respective ca valoare

- **startPoints**: reprezintă o listă în care se pun punctele de start pentru generarea fluxului de mașini.

Metodele din cadrul acestei clase sunt:

- **clear()**: metodă ce șterge datele din attributele clasei. Am implementat acest lucru deoarece de fiecare dată când se face o noua simulare, trebuie șterse datele precedente
- **generateMap(String mapName)**: în această metodă se face citirea din fișierul ce generează harta
- **generateFlow(flowName)**: în această metodă se face citirea din fișierul ce generează fluxul de mașini.

Ca și în metoda generateMap, se citește din fișier cu ajutorul claselor File și Scanner. Secvența de cod este prezentată mai jos:

```
1.  StringBuilder content = new StringBuilder();
2.  try {
3.      File myObj = new File(flowName);
4.      Scanner myReader = new Scanner(myObj);
5.      while (myReader.hasNextLine()) {
6.          String data = myReader.nextLine();
7.          content.append(data);
8.      }
9.      myReader.close();
10. } catch (FileNotFoundException e) {
11.     System.out.println("FILE NOT FOUND!!!");
12.     e.printStackTrace();
13. }
```

După ce se citește conținutul fișierelor, se folosește pachetul org.json.*; ce crează obiecte sau array-uri în format JSON. Aceste obiecte pot fi manipulate mult mai ușor sub formatul JSON. În cazul în care nu s-a introdus o variabilă corectă în fisierul JSON, librăria afișează în consolă excepții cu problema respectivă.

Conversia din string(variabila content) în obiecte JSON este prezentată mai jos:

```
1.  try {
2.      JSONObject json = new JSONObject(content.toString());
3.      this.flowTime = json.getInt("flowTime");
4.      //this.flowTime = Integer.MAX_VALUE;
5.      JSONArray startPointsArray = json.getJSONArray("startPoints");
6. } catch (JSONException e) {
7.     e.printStackTrace();
8. }
9. }
```

- **createCrossroad(JSONObject arrayObject)**: reprezintă metoda ce crează obiectele de tipul intersecție și le pune în HashMap-ul hashMap. Argumentul arrayObject este de tipul clasei JSON și se pot lua date din acesta. Datele ce sunt luate din acest obiect sunt următoarele: numele intersecției, coordonatele x și y a acesteia, și tipul(adică unde sunt poziționate semafoarele). Codul ce face luarea acestor date este reprezentat mai jos:

```
1.  String name = arrayObject.getString("name");
2.  int x = arrayObject.getInt("x");
3.  int y = arrayObject.getInt("y");
```



```
4. String type = (String) arrayObject.get("type");
5.
```

După ce se face citirea acestor date, se crează un HashMap în care se pun intersecțiile specifice (în capitolul 5.3.1.4 se pot observa argumentele constructorului reprezentate de nume, coordonate, timpul și un HashMap) și în funcție de datele din fișier, se adaugă în HashMap-ul respectiv, semaforul și poziția acestuia. Un exemplu de cod ce verifică poziția semaforului din fișier, după care se pune în HashMap-ul principal (cu numele intersecției și clasa respectivă intersecției) este:

```
1. for (int i = 0; i < lightsFromFile.length(); i++) {
2.     JSONObject jsonObject = (JSONObject) lightsFromFile.get(i);
3.     if (type.contains("U")) {
4.         double upTime = jsonObject.getDouble("up");
5.         lights.put('U', new Light('U', upTime));
6.     }
7.     .....
8. }
9. Crossroad crossroad = new Crossroad(name, x, y, type, lights);
10. this.hashMap.put(name, crossroad);
11.
```

- **createRoads(JSONObject arrayObject):** reprezintă metoda ce crează obiectele de tipul Road și le pune în lista. Se iau următoarele date din variabila arrayObject: numele strazii, numele intersecției de unde iese strada, numele intersecției unde intră strada și tipul acesteia. Pe baza acestor date se formează obiectele Road cu datele specifice obiectului descrise în 5.3.1.3. După crearea acestor obiecte, se pun în lista roads și această listă este accesată din pachetul view, adică Clasa Traffic. Un exemplu de cod ce crează un obiect Road cu datele citite din fișier este prezentat mai jos. Al doilea if reprezintă condiția în care utilizatorul a introdus date cu dimensiuni mai mari decât fereastra inițială a aplicației:

```
1. if (t.equals("UD")) {
2.     Road r = new Road(name, distance, startCrossroad.getX() + 7 - 7,
3.         startCrossroad.getY(), 16, auxDistance + 30, Road.Type.UD, endCrossroad, false);
4.     startCrossroad.addRoad(r, 1);
5.     roads.add(r);
6.     if (startCrossroad.getY() + auxDistance > Traffic.getScrollPaneHeight()) {
7.         Traffic.changePanel(Traffic.getScrollPaneWidth(), startCrossroad.getY() +
8.             auxDistance + 100);
9.     }
10. }
```

5.3.3.3. Clasa WriteToFile

Această clasă formează fișierul în care se pun rezultatele simulării. Acest lucru este posibil prin folosirea bibliotecii java.io.*; ce crează o clasă de tip fișier. În acest fișier se pot scrie date indiferent de acestea.

Această clasă nu are constructor, doar trei metode statice ce sunt apelate din alte clase. Metodele sunt următoarele:

- **createFile():** este o metodă ce folosește clasa FileWriter. Această clasă crează un fișier cu numele specificat la declararea obiectului și se scriu date în acesta folosind metoda write. Secțiunea de cod ce face crearea și scrierea în fișier este prezentată mai jos:

```

1.  FileWriter myWriter = new FileWriter("Results.txt");
2.      for (Vehicle car : cars) {
3.          myWriter.write("Car: " + carIndex +
4.              ", distance: " +
5.              String.format("%.02f", (float)car.getDistance() / 1000) + "km" +
6.              ", time: " +
7.              String.format("%.02f", (car.getSimulationStep() * 0.1) / 60) + "min" + "" +
8.              ", steps: " + car.getSimulationStep() + "\n");
9.          carIndex++;
10.     }

```

Tot în această metodă se fac calculele pentru a adapta rezultatele din simulare pentru viața reală. Mai exact, în simulator un pixel reprezintă un metru din viața reală. Când se scrie în fișier, se va scrie sub forma de kilometri așa că se împarte distanța la 1000.

Se consideră că mașinile merg cu 50km/h. În simulator, mașinile se incrementează cu 1.4 pixeli la fiecare pas de simulare (pixelii nu pot avea valori reale, aceasta este valoarea cu care se incrementează la fiecare pas, când se afișează pe ecran se face typecast la această valoare în una întreagă pentru a nu produce erori în program). Am ales valoarea de 1.4 deoarece $5000\text{m}/3600\text{sec} = 14\text{m/s}$. Așadar o mașină face 1.4 m la fiecare 0.1 secunde. Acestea fiind spuse, când se calculează timpul ce se scrie în fișier, se iau numărul de pași de simulare și se înmulțește la 0.1 rezultatul fiind în secunde. De aceea se împarte la valoarea 60 pentru a fi transformat în minute.

Valoarea medie a masinilor se calculează însumând toți timpii și toate distanțele, după care se împarte distanța la timp pentru a obține media.

- **openFile()**: această metodă se apelează atunci când utilizatorul alege să deschidă fișierul după terminarea simulării prin apăsarea unui buton. Pentru a face posibil acest lucru, am folosit clasa Desktop și cu ajutorul metodei open() se deschide fișierul Results.txt. Implementarea acestei metode se află în Anexa 1.

5.3.3.4. Clasa GenerateVehicles

Această clasă este responsabilă pentru generarea vehiculelor la anumite secunde. Am implementat o astfel de clasă pentru a face mai ușoară generarea de mașini atunci când este nevoie în cod și a nu repeta aceeași secvență de cod.

Atributele din aceasta clasa sunt:

- **ONE_SECOND**: are valoarea 100
- **mils**: este valoarea ce se incrementează la fiecare pas de simulare

Clasa nu are un constructor (doar cel implicit), însă este formată următoarea metodă, pe lângă metoda de set:

- **run(double time)**: este metoda ce se apelează atunci când se dorește generarea unui flux de mașini. Această metodă doar verifică dacă s-a ajuns la N secunde. La fiecare pas de simulare se incrementează o variabilă mils. Atunci când această variabilă a ajuns la 100, înseamnă că a trecut o secundă (se verifică dacă a ajuns la 100 deoarece variabilă mils se incrementează o dată la 10 milisecunde).

5.4. Ordinea evenimentelor din aplicație

Pentru a înțelege cum funcționează aplicația, trebuie explicat și ordinea apelării metodelor. Acesta consider că este un lucru important deoarece așa se poate da seama cum a fost dezvoltată aplicația și care este logica acesteia. Chiar dacă am explicat în 4.1.1 care este flow-ul aplicației, mai jos sunt prezentate ordinea metodelor importante.

La rularea aplicației prima metodă apelată este **main(String[] args)**. În această metodă se declară un obiect de tip **Traffic** ce deschide aplicația grafică. **Traffic()** adaugă componentele cu care utilizatorul poate interacționa. **startFrame()** este metoda ce adaugă în panel, componentele ce formează „prima pagină” a aplicației în care se află numele aplicației și o intersecție simplă. **actionPerformed(ActionEvent event)** „așteaptă” datele introduse de utilizator și acțiunile acestuia (ce butoane apasă). **ReadFromFile(String mapName, String flowName)** face citirea din fișier dacă se apasă butonul „Generate Map”. **thread.start()** este apelată dacă apasă butonul de „Start”. La apăsarea butonului de start se produce simularea și se apelează următoarele metode din clasa **Map**: **step()** → **carCollision(float x, float y, Vehicle v)** → **lightCollision(float x, float y, Vehicle v)** → **getNextRoad(Vehicle v)** → **isOnNextRoad(Vehicle v, Road r)** → **changeRoad(Vehicle v, Road r)** → **v.setX()** sau **v.setY()**. Aceste metode fac mașinile să se miște în simulator. **crossroad.run()**: începe rularea fiecărui semafor din simulator. **generateVehicles.run(startPoint.getTime())** generează mașini la N secunde. **map.addCar()** adaugă mașina generată. **frame.repaint()** metodă ce actualizează interfața grafică.

La finalul simulării se apelează **WriteToFile.createFile()** pentru a crea fișierul text și a pune rezultatele în acesta după care **WriteToFile.openFile()** în cazul în care utilizatorul alege să deschidă fișierul din aplicație.

5.5. Instrumente utilizate

Instrumentele utilizate de mine pentru dezvoltarea aplicației sunt:

- mediul de programare **IntelliJ** în care am scris codul. Am ales acest mediu de programare deoarece la toate proiectele de facultate am folosit acest mediu. Mai mult de atât are o interfață grafică user-friendly, face mult mai ușor de realizat scrierea și indexarea codului în limbajul Java. Mai mult de atât se pot vedea mult mai bine și mai clar pachetele și clasele create pentru dezvoltarea aplicației
- limbajul de programare **Java** deoarece este un limbaj orientat pe obiect și este cel mai folosit limbaj de mine. Acest limbaj are sintaxa asemănătoare ca cea din C și clase deja implementate ce sunt foarte utile cum ar fi: **ArrayList**, **Object**, **HashMap**, **Queue** ce m-au ajutat la stocarea și procesarea datelor din aplicație.
- tool-ul pentru dezvoltarea interfețelor grafice **Java Swing** ce are toate componentele necesare pentru dezvoltarea unei astfel de aplicații (forme geometrice, butoane, combo box-uri, text field-uri, etc.)
- tool-urile pentru implementarea diagramelor necesare **StarUML** și site-ul <https://www.diagrameditor.com/>. Acestea oferă toate tipurile de componente de legături între acestea. Diagramele arhitecturale variază foarte mult una de cealaltă prin simboluri, forme geometrice și săgeți. Aceste tool-uri oferă toate componentele necesare pentru realizarea acesteia. Mai mult de atât se pot personaliza formele geometrice după bunul plac.

Capitolul 6. Testare și Validare

În acest capitol se vor prezenta în ce cazuri am testat aplicația și ce rezultate am obținut pentru fiecare caz. Așadar voi prezenta pe o hartă scrisă de mine, trei cazuri de testare și cele trei rezultate obținute pe aceasta. Harta pe care s-a făcut testarea este prezentată în Capitolul 7.

6.1. Cazuri de testare

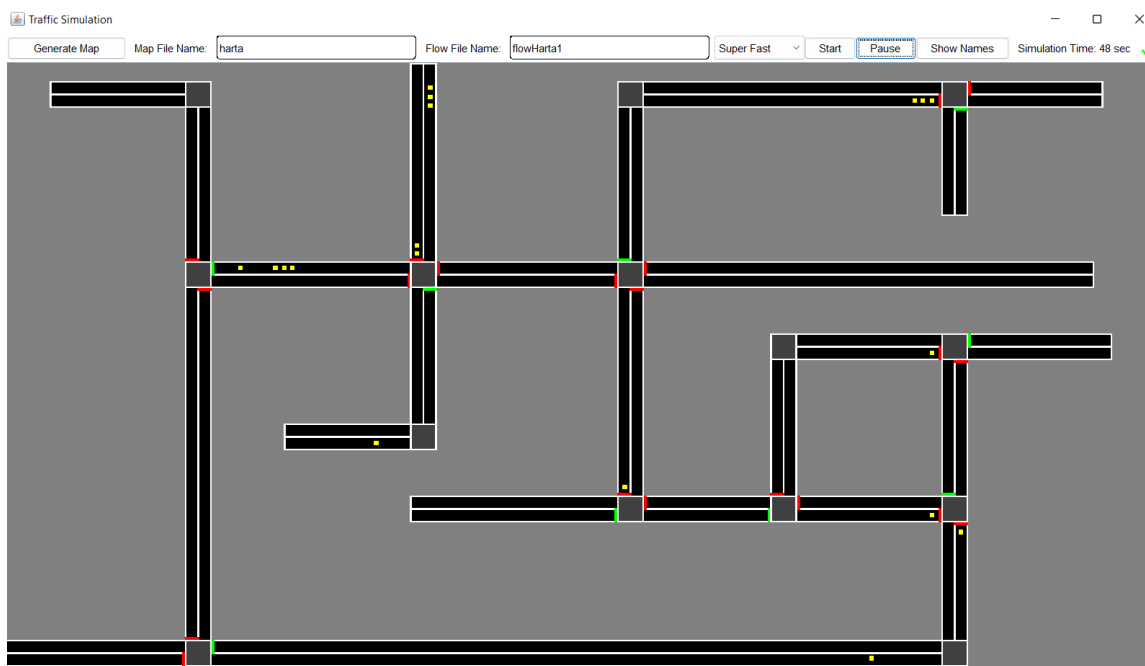
Cazurile de testare oferă o analiză finală a proiectului deoarece cu ajutorul acestora se pot analiza dacă aceste cazuri oferă rezultatele așteptate sau nu. Am implementat trei cazuri de testare

- Harta cu un flux mai rar, adică se generează puține mașini pe două dintre străzile din harta
- Harta cu un flux mediu, adică se generează puține mașini pe mai multe străzi oferind un flux normal de mașini și de generare de rezultate normale
- Harta cu un flux intens, adică pe multe străzi se generează multe mașini pentru a produce blocaje pe diferite străzi

Imaginile prezentate mai jos sunt o vizualizare a intensității traficului după 50 de secunde de simulare. Așadar am oprit simularea la secunda 50 deoarece fiecare flux are setat timpul de simulare de 100 de secunde. După aceasta se poate vizualiza fișierul cu rezultate.

- **Flux rar de generare a mașinilor**

În acest caz de utilizare am ales să generez mașini doar pe două străzi, pe o stradă se generează mașini la trei secunde și pe cealaltă se generează mașini o dată la cinci secunde. Harta pentru acest caz de utilizare poate fi vizualizată în Figură 6.1.

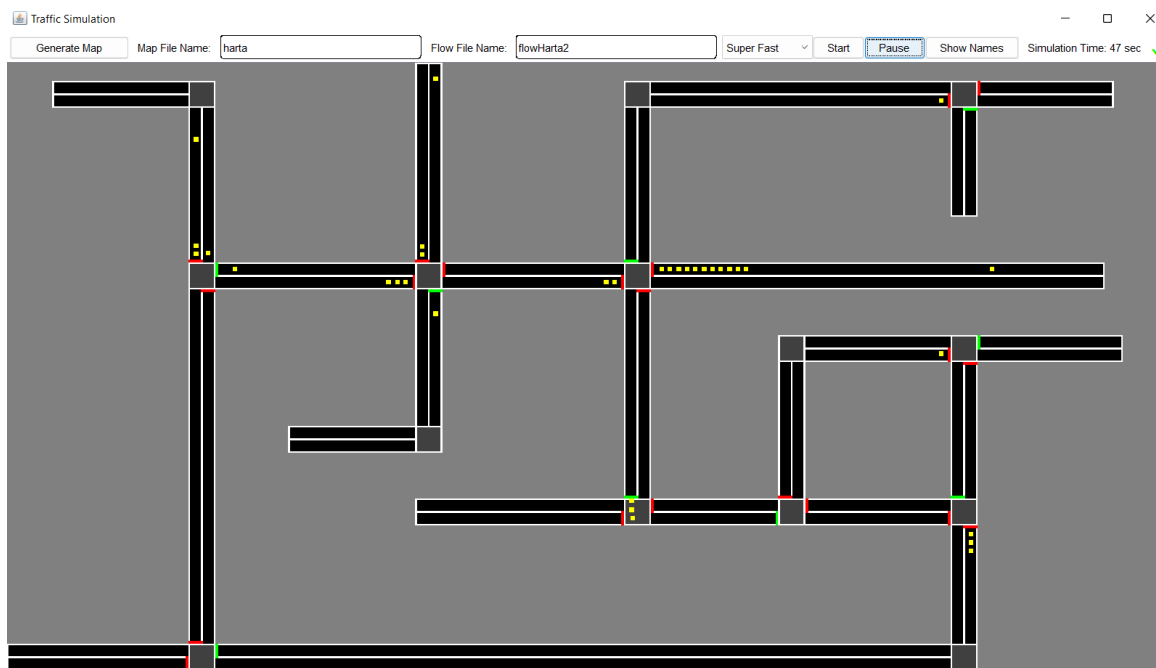


Figură 6.1. Flux rar de generare a mașinilor

După cum se poate observa mai sus, sunt puține mașini generate și intensitatea traficului este intens. Fiind generate puține mașini, se poate observa în următorul subcapitol că sunt puține mașini ce își termina traseul stabilit

- **Flux normal de generare a mașinilor**

În acest caz de utilizare, am scris în fișierul de generare de fluxuri de mașini un scenariu în care simularea produce rezultate normale. Eu am folosit acest scenariu în dezvoltarea aplicației. Un exemplu ce poate fi vizualizat pentru acest scenariu de simulare este prezentat în Figură 6.2.



Figură 6.2. Flux normal de generare a mașinilor

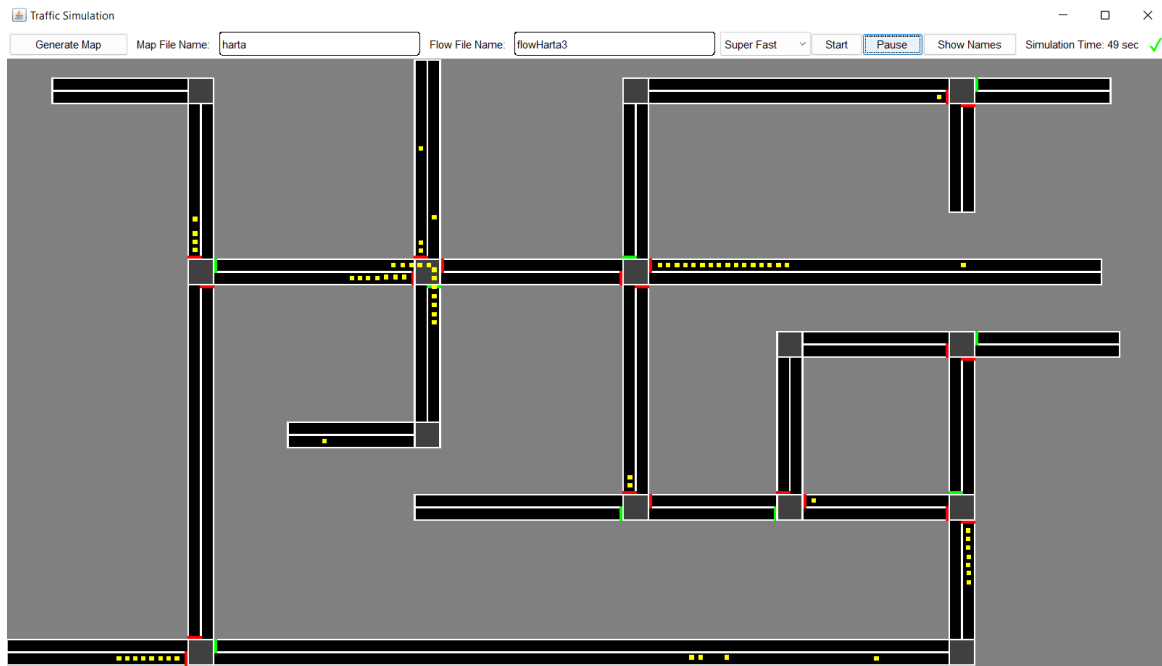
După cum se poate observa, există un număr mediu de mașini și nu se produce aglomerație pe anumite străzi, indiferent de distanțele acestora. În acest caz de simulare se consideră dacă o stradă are o distanță mică, se vor genera mașini la un timp mai mare pentru a nu produce ambuteiaj.

- **Flux intens de generare a mașinilor**

Acest caz de simulare este cel mai intens pentru un caz de simulare deoarece aplicația are de procesat un număr mare de agenți. Mai mult de atât, în acest caz de simulare se pot produce blocaje pe o anumită stradă din cauză că pe acea stradă sunt prea multe mașini și o mașină „va intra” în alta. Aplicația informează utilizatorul unde se produce accidentul cu numele străzii respective.

Acest caz de simulare poate fi vizualizat în Figură 6.3. În figură se poate observa evoluția mașinilor la jumătatea timpului de simulare și după cum se poate observa sunt multe mașini ce așteaptă la semafoare.

După câteva rulări ale aceluiași scenariu sau o dată la câteva rulări se generează mesajul că sunt prea multe mașini pe aceeași stradă, astfel există posibilitatea producerii unui accident.



Figură 6.3. Flux intens de generare a mașinilor

Un alt caz de testare este atunci când se folosește aceeași hartă și același flux, singura caracteristică modificată este schimbarea timpilor semafoarelor. Acest lucru face posibilă optimizarea timpilor semafoarelor pentru un rezultat cât mai bun.

Așadar, în acest caz de testare s-a folosit harta de mai sus și fluxul normal de generare de mașini. În fișierul JSON am modificat fiecare timp de la fiecare intersecție adăugând 5 secunde în plus la fiecare. Rezultatele sunt cu siguranță diferite deoarece mașinile stând mai mult la semafor, viteza medie din întreaga simulare va scădea de asemenea. Rezultatele sunt afișate în capitolul 6.2.

6.2. Rezultate obținute

Rezultatele obținute pot fi vizualizate și analizate în fișierul generat la finalul simulării, și anume Results.txt. Rezultatele generate diferă de la caz la caz, adică pentru un caz de simulare sunt mai puține mașini ce își termină traseul și în alt caz sunt mai multe mașini ce își termina traseul.

O alta valoare rezultată după terminarea simulării este viteza medie. Această valoare se obține prin adunarea tuturor distanțelor și timpilor generate de fiecare mașină. După adunarea acestora se împart aceste valori obținându-se viteza medie.

- **Fux rar de generare de mașini**

În acest caz de utilizare am ales să generez mașini doar pe două străzi, pe o stradă se generează mașini la trei secunde și pe cealaltă se generează mașini o dată la cinci secunde. Harta pentru acest caz de utilizare poate fi vizualizată în Figură 6.4.

```
File Edit View
Car: 8, distance: 0.54km, time: 2.84min, steps: 1702
Car: 9, distance: 0.54km, time: 2.18min, steps: 1310
Car: 10, distance: 0.54km, time: 1.53min, steps: 916
Car: 11, distance: 0.82km, time: 1.55min, steps: 929
Car: 12, distance: 1.09km, time: 2.53min, steps: 1515
Car: 13, distance: 0.54km, time: 4.14min, steps: 2486
Car: 14, distance: 0.54km, time: 2.83min, steps: 1698
Car: 15, distance: 0.92km, time: 2.64min, steps: 1586
Car: 16, distance: 0.54km, time: 0.68min, steps: 407
Car: 17, distance: 1.13km, time: 2.75min, steps: 1652
Car: 18, distance: 1.26km, time: 9.06min, steps: 5437
Car: 19, distance: 0.92km, time: 1.79min, steps: 1071
Car: 20, distance: 1.19km, time: 6.42min, steps: 3850
Car: 21, distance: 1.19km, time: 3.76min, steps: 2255
Car: 22, distance: 0.54km, time: 3.33min, steps: 1997
Car: 23, distance: 0.54km, time: 0.64min, steps: 385
Car: 24, distance: 1.09km, time: 3.33min, steps: 1996
Car: 25, distance: 1.04km, time: 9.92min, steps: 5954
Car: 26, distance: 0.94km, time: 5.70min, steps: 3418
Car: 27, distance: 0.95km, time: 6.41min, steps: 3843
Car: 28, distance: 0.95km, time: 3.10min, steps: 1859
Car: 29, distance: 0.54km, time: 3.97min, steps: 2382
Car: 30, distance: 0.54km, time: 3.31min, steps: 1987
Car: 31, distance: 0.54km, time: 2.00min, steps: 1199
Car: 32, distance: 1.23km, time: 7.66min, steps: 4596
Car: 33, distance: 1.13km, time: 3.43min, steps: 2055
Car: 34, distance: 1.16km, time: 4.81min, steps: 2883
Car: 35, distance: 0.92km, time: 5.09min, steps: 3055
Car: 36, distance: 1.04km, time: 3.20min, steps: 1921
Car: 37, distance: 1.23km, time: 2.14min, steps: 1282
Car: 38, distance: 0.92km, time: 1.15min, steps: 689
Car: 39, distance: 2.94km, time: 14.32min, steps: 8591
Ln 22, Col 54 100% Unix (LF) UTF-8
```

Figură 6.4. Rezultate pentru flux rar de generare de mașini

După cum se poate observa mai sus, sunt puține mașini generate și intensitatea traficului este intens. Mai exact 39 de mașini în 100 de secunde, fiind puține pentru harta descrisă în subcapitolul precedent.

Pentru acest caz de utilizare s-a generat o viteză medie de 44 km/h.

- **Flux normal de generare de mașini**

Pentru acest caz de simulare s-au produs rezultate convenabile pentru un caz de utilizare normal. Mai exact în 100 de secunde, 74 de mașini și-au terminat traseul pe harta din cazul de utilizare din subcapitolul precedent.

Rezultatele pentru acest caz de simulare pot fi vizualizate în Figură 6.5.

Pentru acest caz de utilizare s-a generat o viteză medie de 41 km/h.

```

File Edit View
Car: 43, distance: 2.11km, time: 7.59min, steps: 4552
Car: 44, distance: 0.82km, time: 1.61min, steps: 965
Car: 45, distance: 0.92km, time: 3.40min, steps: 2040
Car: 46, distance: 0.92km, time: 2.83min, steps: 1696
Car: 47, distance: 0.97km, time: 3.12min, steps: 1870
Car: 48, distance: 0.97km, time: 1.96min, steps: 1176
Car: 49, distance: 0.54km, time: 2.75min, steps: 1652
Car: 50, distance: 0.54km, time: 1.03min, steps: 620
Car: 51, distance: 0.82km, time: 3.66min, steps: 2197
Car: 52, distance: 0.82km, time: 0.98min, steps: 586
Car: 53, distance: 0.54km, time: 0.64min, steps: 385
Car: 54, distance: 0.54km, time: 0.64min, steps: 385
Car: 55, distance: 0.95km, time: 6.78min, steps: 4067
Car: 56, distance: 0.95km, time: 4.45min, steps: 2669
Car: 57, distance: 1.04km, time: 10.09min, steps: 6051
Car: 58, distance: 1.05km, time: 6.66min, steps: 3998
Car: 59, distance: 0.97km, time: 3.88min, steps: 2325
Car: 60, distance: 3.71km, time: 10.50min, steps: 6299
Car: 61, distance: 0.54km, time: 3.24min, steps: 1941
Car: 62, distance: 0.54km, time: 2.09min, steps: 1253
Car: 63, distance: 0.54km, time: 0.94min, steps: 566
Car: 64, distance: 3.23km, time: 13.18min, steps: 7906
Car: 65, distance: 1.51km, time: 8.74min, steps: 5244
Car: 66, distance: 1.13km, time: 7.29min, steps: 4373
Car: 67, distance: 1.26km, time: 9.47min, steps: 5679
Car: 68, distance: 1.13km, time: 4.38min, steps: 2628
Car: 69, distance: 1.26km, time: 8.31min, steps: 4985
Car: 70, distance: 1.26km, time: 7.16min, steps: 4293
Car: 71, distance: 0.92km, time: 4.28min, steps: 2568
Car: 72, distance: 1.04km, time: 2.06min, steps: 1236
Car: 73, distance: 1.26km, time: 5.23min, steps: 3138
Car: 74, distance: 2.89km, time: 14.57min, steps: 8744
Ln 1, Col 1 100% Unix (LF) UTF-8

```

Figură 6.5. Rezultate pentru flux normal de generare de mașini

- **Flux intens de generare de mașini**

Fiind un scenariu intens în care se generează multe mașini, rezultatele nu pot fi vizualizate de fiecare dată pentru acest caz. Atunci când un scenariu de simulare cu un trafic intens ajunge la bun sfârșit, se poate observa în fișierul Results.txt un număr mare de mașini.

Un alt fenomen ce se poate observa este acela ca majoritatea timpilor sunt mai mari decât la celelalte cazuri de utilizare.

Pentru acest caz de utilizare s-a generat o viteză medie de 38 km/h. Rezultatele pentru acest caz de simulare pot fi vizualizate în Figură 6.6.

```

File Edit View
Car: 108, distance: 0.82km, time: 2.29min, steps: 1374
Car: 109, distance: 0.82km, time: 1.93min, steps: 1160
Car: 110, distance: 1.33km, time: 7.51min, steps: 4504
Car: 111, distance: 0.54km, time: 3.90min, steps: 2341
Car: 112, distance: 0.54km, time: 3.19min, steps: 1913
Car: 113, distance: 1.18km, time: 5.07min, steps: 3044
Car: 114, distance: 1.23km, time: 9.77min, steps: 5860
Car: 115, distance: 1.23km, time: 9.41min, steps: 5646
Car: 116, distance: 1.23km, time: 8.32min, steps: 4993
Car: 117, distance: 1.51km, time: 7.53min, steps: 4518
Car: 118, distance: 1.54km, time: 7.22min, steps: 4334
Car: 119, distance: 0.92km, time: 5.09min, steps: 3052
Car: 120, distance: 1.54km, time: 6.87min, steps: 4119
Car: 121, distance: 1.16km, time: 7.38min, steps: 4430
Car: 122, distance: 1.16km, time: 5.55min, steps: 3332
Car: 123, distance: 0.92km, time: 4.02min, steps: 2409
Car: 124, distance: 1.16km, time: 5.20min, steps: 3119
Car: 125, distance: 0.92km, time: 2.59min, steps: 1554
Car: 126, distance: 1.05km, time: 3.07min, steps: 1843
Car: 127, distance: 1.05km, time: 1.65min, steps: 987
Car: 128, distance: 1.79km, time: 12.99min, steps: 7791
Car: 129, distance: 2.36km, time: 13.51min, steps: 8105
Car: 130, distance: 1.26km, time: 5.11min, steps: 3063
Car: 131, distance: 2.36km, time: 13.15min, steps: 7890
Car: 132, distance: 1.26km, time: 3.68min, steps: 2208
Car: 133, distance: 0.92km, time: 1.15min, steps: 689
Car: 134, distance: 1.09km, time: 6.86min, steps: 4116
Car: 135, distance: 0.82km, time: 4.08min, steps: 2448
Car: 136, distance: 0.82km, time: 3.00min, steps: 1800
Car: 137, distance: 1.19km, time: 6.57min, steps: 3941
Car: 138, distance: 1.05km, time: 1.27min, steps: 764
Car: 139, distance: 0.92km, time: 3.79min, steps: 2276
Ln 116, Col 55 100% Unix (LF) UTF-8

```

Figură 6.6. Rezultate pentru flux intens de generare de mașini

Rezultatele obținute pentru cazul de testare în care s-a adăugat la fiecare semafor din simulator 5 secunde în plus sunt următoarele.

Fișierul results se poate vizualiza în Figură 6.7.

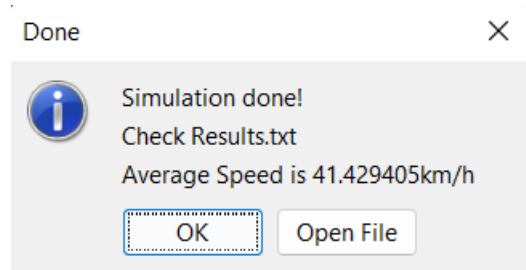
```

File Edit View
Car: 27, distance: 0.54km, time: 1.59min, steps: 955
Car: 28, distance: 0.54km, time: 1.02min, steps: 611
Car: 29, distance: 0.95km, time: 8.43min, steps: 5056
Car: 30, distance: 0.95km, time: 6.13min, steps: 3680
Car: 31, distance: 0.95km, time: 2.69min, steps: 1616
Car: 32, distance: 0.54km, time: 0.64min, steps: 385
Car: 33, distance: 3.01km, time: 10.50min, steps: 6300
Car: 34, distance: 0.54km, time: 0.64min, steps: 385
Car: 35, distance: 0.82km, time: 9.37min, steps: 5624
Car: 36, distance: 0.82km, time: 3.56min, steps: 2135
Car: 37, distance: 0.82km, time: 2.99min, steps: 1791
Car: 38, distance: 1.46km, time: 11.39min, steps: 6831
Car: 39, distance: 2.33km, time: 6.99min, steps: 4193
Car: 40, distance: 1.16km, time: 8.97min, steps: 5379
Car: 41, distance: 1.16km, time: 7.22min, steps: 4333
Car: 42, distance: 1.18km, time: 4.48min, steps: 2686
Car: 43, distance: 1.18km, time: 3.90min, steps: 2342
Car: 44, distance: 1.21km, time: 5.67min, steps: 3404
Car: 45, distance: 0.54km, time: 7.12min, steps: 4274
Car: 46, distance: 1.21km, time: 2.81min, steps: 1684
Car: 47, distance: 1.21km, time: 1.66min, steps: 996
Car: 48, distance: 0.54km, time: 1.96min, steps: 1178
Car: 49, distance: 0.94km, time: 7.04min, steps: 4225
Car: 50, distance: 0.99km, time: 10.15min, steps: 6087
Car: 51, distance: 0.99km, time: 3.78min, steps: 2270
Car: 52, distance: 0.99km, time: 2.63min, steps: 1575
Car: 53, distance: 0.92km, time: 5.30min, steps: 3179
Car: 54, distance: 0.92km, time: 4.15min, steps: 2491
Car: 55, distance: 0.99km, time: 1.17min, steps: 699
Car: 56, distance: 0.54km, time: 0.68min, steps: 407
Ln 1, Col 1 100% Unix (LF) UTF-8
    
```

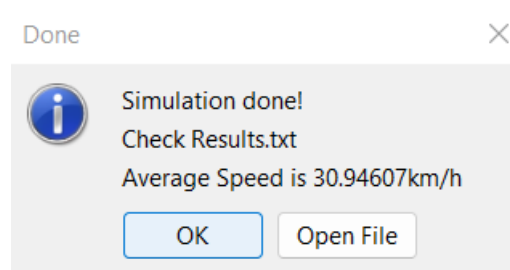
Figură 6.7. Rezultate când s-a adăugat secunde în plus la semafoare

După cum se poate observa, pentru aceeași hartă cu un flux normal de generare a mașinilor, numărul total al mașinilor este mai mic cu aproximativ 20 30 de mașini.

O altă diferență ce se poate observa este viteza medie finală după terminarea simulării. După cum se poate observa în Figură 6.8 și Figură 6.89 exista o diferență de aproximativ 10km/h dintre primul scenariu(înainte de modificarea semafoarelor) și următorul(timpul semafoarelor a fost prelungit).



Figură 6.8. Scenariul fără 5 secunde



Figură 6.9. Scenariul cu 5 secunde adăugate

În primul scenariu s-au afișat rezultate în jurul valorilor 38-42km/k, iar în al doilea scenariu s-au afișat valori între 30-34 km/h. Așadar, cu cât mașinile stau mai mult la semafoare, viteza medie scade considerabil reducând eficiența.

Capitolul 7. Manual de Instalare și Utilizare

În acest capitol se vor prezenta pașii de instalare și utilizare ai aplicației. Această parte este importantă pentru utilizator deoarece se explică funcționalitățile interfeței grafice și pașii din procesul de instalare al aplicației.

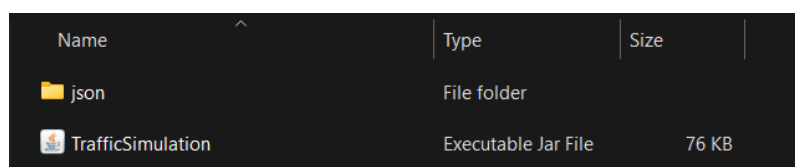
7.1. Instalare

Aplicația este arhivată, iar utilizatorul trebuie să o dezarhiveze prin intermediul unei aplicații care oferă acest serviciu. Un exemplu de aplicație este WinRAR(Figură 7.1).



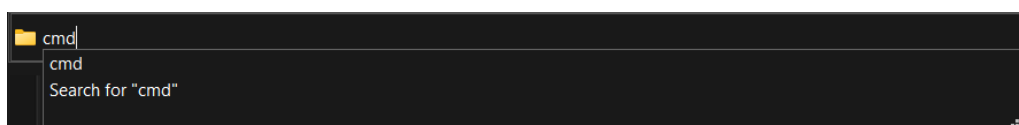
Figură 7.1. Arhiva aplicației

După ce se dezarhivează, va apărea directorul TrafficSimulation în care se află aplicația .jar și directorul json ce conține fișierele de configurare a hărților și fluxul mașinilor. (Figură 7.2)



Figură 7.2. Directorul TrafficSimulation

Pentru a rula aplicația, se deschide un terminal(sau Command Prompt pentru windows). Pentru linux se apasă click dreapta în director și se selectează „Open Terminal”. Pentru a deschide un terminal în Windows se scrie „cmd” în bara de directoare(Figură 7.3) și se apasă Enter. Acest lucru se mai poate face deschizând un terminal din bara de start și cu ajutorul comenzii „cd” se specifică calea spre directorul aplicației.

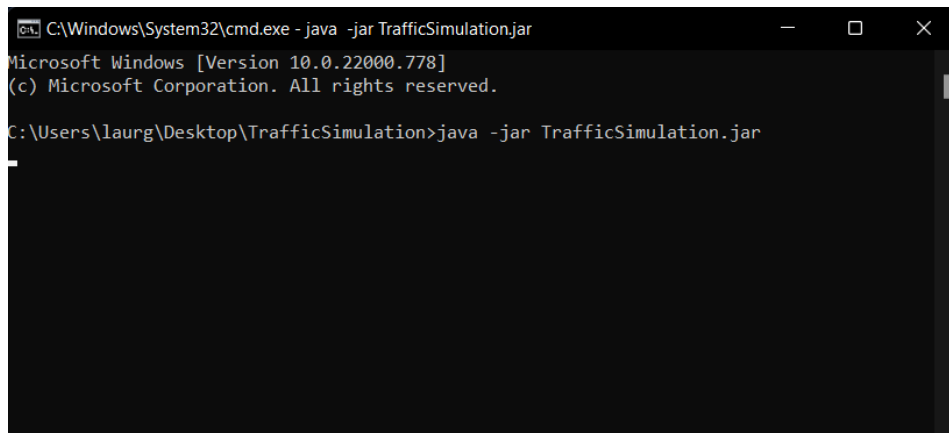


Figură 7.3. Deschiderea terminalului în Windows

O dată deschis terminalul se introduce următoarea comandă ce deschide aplicația(în terminal se pot vedea și excepțiile sau problemele ce apar în timpul execuției. De exemplu, dacă o cheie din fișierul json nu a fost scrisă corect). Acest lucru se poate observa în Figură 7.4.

java -jar TrafficSimulation.jar

Mediul de dezvoltare Java trebuie să fie instalat pe mașina utilizatorului(The Java Development Kit) pentru a putea pune în execuție această aplicație.

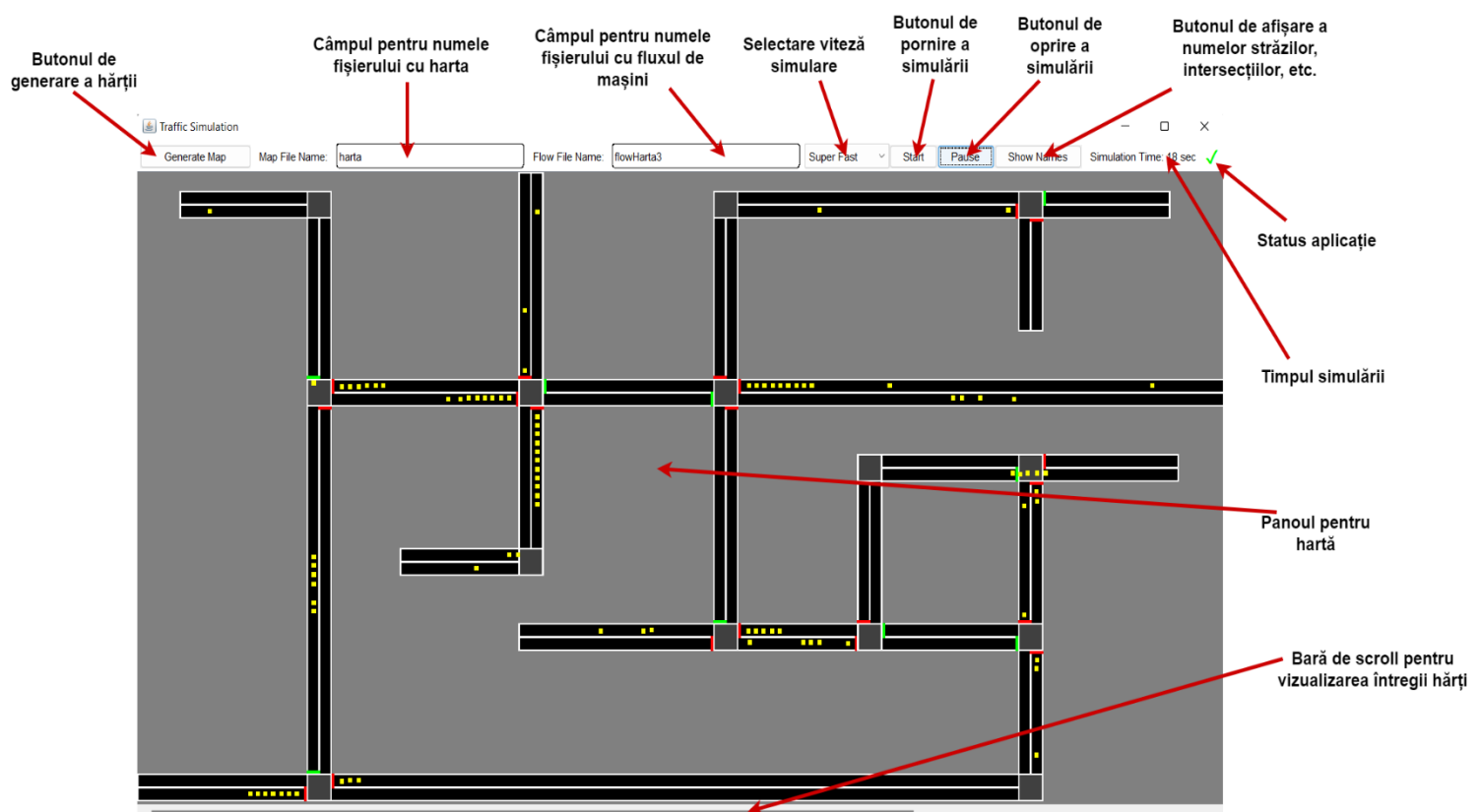


Figură 7.4. Pornirea aplicației

După ce se apasă Enter se va deschide interfața grafică.

7.2. Interfața grafică

În Figură 7.5 se poate observa harta generată după ce utilizatorul a introdus numele fișierului hărții și numele fișierului ce generează fluxul de mașini. Mai jos se pot observa componentele interfeței grafice.



Figură 7.5. Interfața grafică a aplicației

7.3. Utilizarea aplicației

Înainte de a explica pașii ce trebuie făcuți pentru utilizarea aplicației, un lucru important este acela de a înțelege ce se scrie în fișierele JSON. Pentru a ușura înțelegerea utilizatorului, am scris în directoarele maps și flows două fișiere ce reprezintă un template pentru informațiile ce trebuie introduse în fișierul JSON, mai exact fișierele JSONStructure.json (între caracterele „< >”) pentru a crea harta și un flux nou. Utilizatorul poate da copy-paste din fișierele JSONSample.json pentru generarea mai ușoară a hărții sau a fluxului de mașini.

Așadar, câmpurile din fișierele JSON sunt următoarele

- Pentru fișierele din directorul maps (Figură 7.6)

```
{
  "Name": "<Numele hartii>",
  "Crossroads": [
    {
      "name": "<Numele intersectiei>",
      "x": <Coordonata X a intersectiei>,
      "y": <Coordonata Y a intersectiei>,
      "type": "<Unde sunt semafoarele(U,L,R,D)>",
      "lightTimes": [
        {
          "up": <Numarul de secunde pentru semaforul U>,
          "down": <Numarul de secunde pentru semaforul D>,
          "left": <Numarul de secunde pentru semaforul L>,
          "right": <Numarul de secunde pentru semaforul R>
        }
      ]
      // timpii inseamna cat de mult tine culoarea verde pentru fiecare semafor
    }
  ],
  // se pot declara mai multe intersectii separate prin virgula
  "Roads": [
    {
      "name": "<Numele strazii>",
      "startCrossroad": "<Numele intersectiei de unde porneste drumul>",
      "endCrossroad": "<Numele intersectiei in care intra drumul>",
      "type": "<Tipul drumului(LR,RL,UD,DU)>",
      "distance": <Distanța drumului(in metri)>,
      // !!! distanta se specifica in cazul in care una dintre valorile
      // startCrossroad sau endCrossroad este nula
    }
  ],
  // se pot declara mai multe strazi separate prin virgula
}
```

Figură 7.6. Structura pentru fișierul JSON de generare harta

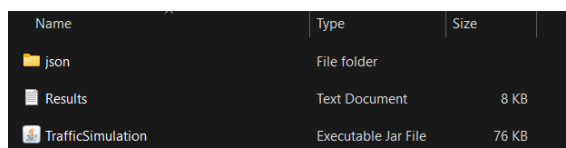
- Pentru fișierele din directorul flows (Figură 7.7)

```
{
  "flowTime": <Numarul de secunde pentru simulare>,
  "startPoints": [
    {
      "roadName": "<Numele strazii pe care se genereaza masini>",
      "carGenerationTime": <Numarul de secunde pentru generarea masinilor>
    }
  ],
  // se pot declara mai multe strazi pe care se genereaza separate prin virgula
}
```

Figură 7.7. Structura pentru fișierul JSON de generare flux

După introducerea datelor în fișierele JSON, utilizatorul deschide aplicația cu comanda de mai sus și va avea de făcut **următori pași**:

1. Introducere nume fișier hartă
2. Introducere nume fișier flux de mașini
3. Apăsarea butonului „Generate Map”
4. După generarea hărții se apasă butonul de „Start”
5. La finalul simulării acesta poate vizualiza fișierul Results.txt generat de aplicație(Figură 7.8). Acest fișier poate fi deschis și din intermediul aplicației



Name	Type	Size
json	File folder	
Results	Text Document	8 KB
TrafficSimulation	Executable Jar File	76 KB

Figură 7.8. Fișierele după terminarea simulării

Capitolul 8. Concluzii

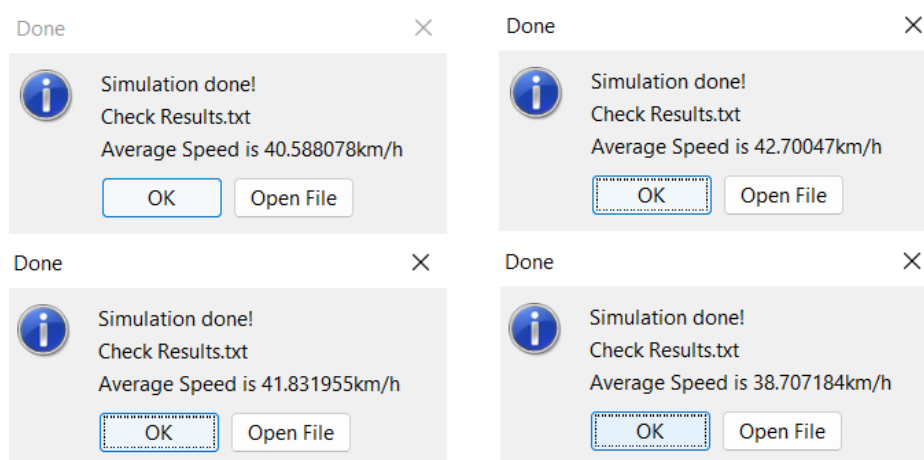
8.1. Scurt rezumat și rezultate obținute.

Dezvoltarea unui astfel de proiect oferă beneficii când vine vorba de simularea și dezvoltarea unui scenariu de trafic urban sau rural. Infrastructura orașului va fi proiectată în urma rezultatelor obținute de către aplicație. În aplicație se vor încărca fișiere cu diferite topologii ale străzilor, iar aplicația va simula traficul generat de mașini pe infrastructura respectivă.

Metoda abordată este una simplă și ușor de înțeles, atât în privința implementării, cât și în privința utilizării și înțelegerii principiilor de funcționare ale aplicației. S-a dezvoltat un proiect robust prin faptul că nu ocupă multă memorie și folosește resurse minime pentru a putea fi utilizată. În general, într-o aplicație de acest gen cu componente care sunt procesate în paralel este nevoie de un număr mare de thread-uri. Pentru a executa aplicații cu multiple thread-uri, este nevoie de un sistem complex și costisitor, prin urmare am ales implementarea unei aplicații în care toate calculele și evoluțiile agenților din simulare să fie făcut pe un singur thread ce rulează la fiecare 10 milisecunde. Am ales acest interval de timp deoarece am considerat că este suficient pentru a actualiza starea agenților, chiar dacă la simulare participă un număr mare de agenți.

Partea de implementare este logică și ușor de înțeles deoarece am luat fiecare caz de utilizare și componentă din flow-ul aplicației(4.1.1) în ordine de sus în jos și nu am lucrat la mai multe funcționalități în același timp.

Rezultatele obținute sunt cele așteptate. Am verificat acest lucru prin analizarea valorilor vitezelor obținute în funcție de parametrii setați înainte de simulare. Am însumat valorile vitezelor și le-am împărțit la distanțe și timp, astfel am obținut medii în următorul interval: 38km/h – 45km/h. Aceste rezultate variază în funcție de hartă și de numărul de mașini. În Figură 8.1 se pot observa valorile obținute după executarea unor cazuri aleatorii. Am ales să calculez această medie pentru a verifica dacă aplicația se adaptează corespunzător valorilor atribuite în simulator, spre deosebire de datele din viața reală. Calcularea acestei valori și exactitatea acesteia este importantă deoarece poate determina un consum mediu pentru scenariul respectiv.



Figură 8.1. Rezultate

La partea de implementare grafica, am ales sa fie cat mai simplu și ușor de înțeles de aceea am folosit forme geometrice simple, pentru a nu influenta într-un oarecare fel procesarea rezultatelor. Bibliotecă folosită pentru generarea părții grafice(Java Swing) poate folosi un număr mai mare de resurse în funcție de complexitatea acestora deoarece aceasta folosește thread-uri pentru anumite componente. Acest lucru poate duce la procesarea greșită a rezultatelor.

8.2. Dezvoltări ulterioare

În viața reală sunt multe scenarii și puncte de luat în vedere atunci când vine vorba de trafic. Câteva dintre exemple sunt apariția de obstacole, sunt multiple benzi pe același sens, infrastructura pietonală, benzi speciale ce au o viteză maximă specifică anumitor tipuri de vehicule și multe altele.

Aceste scenarii pot fi implementate într-o aplicație de acest gen, dar pentru fiecare dintre acestea, trebuie implementate alte cazuri de utilizare și implementare.

Când vine vorba de implementare se pot adăuga următoarele dezvoltări ulterioare:

- adăugarea mai multor benzi și semafoare pe același sens deoarece se pot testa mai multe scenarii
- adăugarea de obstacole pe stradă și pe care mașinile să le ocolească
- implementarea unor străzi curbate
- implementarea mai multor tipuri de vehicule(camioane, biciclete, etc.)
- implementare structura pietonală

Bibliografie

- [1] N. Pettersson, „Modelling and Simulation of Heterogeneous Traffic,” Gothenburg, Sweden, 2020, pp. 5-15.
- [2] C. Mallikarjuna și R. Rao, „Heterogeneous traffic flow modelling: a complete methodology,” *Transportmetrica*, vol. 7, pp. 321-345, 2011.
- [3] J. Stuster, „Aggressive driving enforcement: Evaluation of two demonstration programs,” Martie 2004. [Interactiv]. Available: <https://rosap.ntl.bts.gov/view/dot/1718>.
- [4] SAE International, „Operational definitions of driving performance measures and statistics,” International Standards Organization, 2013.
- [5] National Research Council, „Highway capacity manual,” 2000. [Interactiv]. Available: https://sjnavarro.files.wordpress.com/2008/08/highway_capacital_manual.pdf.
- [6] The University of Memphis, „Traffic flow characteristics,” Septembrie 2019. [Interactiv]. Available: http://www.ce.memphis.edu/4162/L1_Traffic_Flow_Parameters.pdf.
- [7] SUMO, „User Documentation,” [Interactiv]. Available: <https://sumo.dlr.de/docs/index.html>. [Accesat 2022].
- [8] T. Fotherby, „Visual Traffic Simulation,” 2002. [Interactiv]. Available: <https://www.tomfotherby.com/Websites/VISSIM/>.
- [9] M. Treiber, „traffic-simulation,” 2015. [Interactiv]. Available: <https://traffic-simulation.de/>. [Accesat 2022].
- [10] javaTpoint, „Agent Environment in AI,” [Interactiv]. Available: <https://www.javatpoint.com/types-of-ai-agents>. [Accesat 2021].
- [11] CUEMATH, „Distance Between Two Points,” [Interactiv]. Available: <https://www.cuemath.com/geometry/distance-between-two-points/>.
- [12] W3Schools, „Java Threads,” Refsnes Data, [Interactiv]. Available: https://www.w3schools.com/java/java_threads.asp.
- [13] StackOverflow, „What is a "thread" (really)?,” Stack Exchange Inc, [Interactiv]. Available: <https://stackoverflow.com/questions/5201852/what-is-a-thread-really>.
- [14] JavaTpoint, „Java Tutorial,” JavaTpoint, [Interactiv]. Available: <https://www.javatpoint.com/java-tutorial>.
- [15] B. Cole, D. Wood, M. Loy, J. Elliott, R. Eckstein și B. Cole, Java™ Swing, 2nd Edition, Noiembrie: O'Reilly, 2002.

- [16] Rafael Hernandez, „The Model View Controller Pattern – MVC Architecture and Frameworks Explained,” freeCodeCamp, [Interactiv]. Available: <https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvc-architecture-and-frameworks-explained/>. [Accesat 2022].
- [17] E. Gamma, R. Helm, R. Johnson și J. Vlissides, „What Is a Design Pattern?,” în *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994, pp. 1-29.
- [18] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Paperback, 2003.
- [19] G. Ion-Augustin, „Programare Orientată pe Obiecte,” [Interactiv]. Available: http://users.utcluj.ro/~igiosan/teaching_poo.html.

Anexa 1

1. Metoda run din cadrul thread-ului

```

1.  @Override
2.      public void run() {
3.          while (running && ok) {
4.              if (countDown > 0) {
5.                  mils++;
6.                  map.step();
7.                  for (Crossroad crossroad : crossroads) {
8.                      crossroad.run();
9.                  }
10.                 if (countDown > 0) {
11.                     for (StartPoints startPoint : startPoints) {
12.                         if (generateVehicles.run(startPoint.getTime())) {
13.                             try {
14.                                 Car car = new Car(startPoint.getRoad(),
15.                                                         startPoint.getRoad().getStartX(),
16.                                                         startPoint.getRoad().getStartY());
17.                                 car.setDistance(car.getRoad().getDistance());
18.                                 map.addCar(car);
19.                             } catch (IllegalArgumentException e) {
20.                                 System.out.println("Nu exista puncte de start
21. pentru mașini");
22.                                 System.exit(0);
23.                             }
24.                         }
25.                     }
26.                     this.timer();
27.                     frame.repaint();
28.                     try {
29.                         Thread.sleep(simulationType);
30.                     } catch (Exception e) {
31.                         e.printStackTrace();
32.                     }
33.                 } else {
34.                     frame.repaint();
35.                     running = false;
36.                     ok = false;
37.                     WriteToFile.createFile();
38.                     String[] buttons = {"OK", "Open File"};
39.
40.                     int result = JOptionPane.showOptionDialog(frame, "Simulation
41. done!\nCheck Results.txt\nAverage Speed is " + WriteToFile.getAverageSpeed() +
42. "km/h",
43. "Done", JOptionPane.YES_NO_CANCEL_OPTION,
44. JOptionPane.INFORMATION_MESSAGE, null, buttons, buttons[0]);
45.                     if (result == 1) {
46.                         WriteToFile.openFile();
47.                     }
48.                 }
49.             }
50.         }

```

2. Metoda ce urmărește acțiunile butoanelor:

```

1. @Override
2.     public void actionPerformed(ActionEvent event) {
3.         if (event.getSource().equals(getMap)) {
4.             map.clear();
5.             mils = 0;
6.             generateVehicles.setMils(0);
7.             running = false;
8.             setStatus(true);
9.             comboBox.setSelectedIndex(1);
10.
11.             if (mapFile.getText().equals("")) {
12.                 setStatus(false);
13.                 JOptionPane.showMessageDialog(null, "Map not selected");
14.             } else if (flowFile.getText().equals("")) {
15.                 setStatus(false);
16.                 JOptionPane.showMessageDialog(null, "Flow not selected");
17.             } else {
18.                 System.out.println("Map: " + mapFile.getText());
19.                 System.out.println("Flow: " + flowFile.getText());
20.                 readFromFile = new ReadFromFile("json/maps/" + mapFile.getText()
+ ".json",
21.                                     "json/flows/" + flowFile.getText() + ".json");
22.
23.                 if (ok) {
24.                     idle = false;
25.                     this.countDown = readFromFile.getFlowTime();
26.                     this.countDownLabel.setText("Simulation Time: " + countDown +
" sec");
27.
28.                     this.crossroads = readFromFile.getCrossroads();
29.                     for (Crossroad crossroad : crossroads) {
30.                         map.addCrossroad(crossroad);
31.                     }
32.                     this.roads = readFromFile.getRoads();
33.                     for (Road road : roads) {
34.                         map.addRoad(road);
35.                     }
36.                     this.startPoints = readFromFile.getStartPoints();
37.                     frame.repaint();
38.                     frame.setSize(new Dimension(WIDTH - 1, HEIGHT - 1));
39.                     frame.setResizable(true);
40.                 } else {
41.                     startFrame();
42.                 }
43.             }
44.         }
45.         if (event.getSource().equals(start)) {
46.             if (!running && ok) {
47.                 running = true;
48.                 thread = new Thread(this);
49.                 thread.start();
50.             }
51.         }
52.         if (event.getSource().equals(stop)) {
53.             running = false;
54.         }
55.         if (event.getSource().equals(showNamesButton)) {
56.             if (!showNames) {
57.                 showNamesButton.setText("Hide Names");
58.                 showNames = true;
59.                 frame.repaint();
60.             } else {
61.                 showNamesButton.setText("Show Names");
62.                 showNames = false;
63.                 frame.repaint();

```

```

64.         }
65.     }
66. }
67.

```

3. Metodă ce generează fișierul Results.txt

```

1. public static void createFile() {
2.     int carIndex = 0;
3.     try {
4.         FileWriter myWriter = new FileWriter("Results.txt");
5.         for (Vehicle car : cars) {
6.             myWriter.write("Car: " + carIndex +
7.                 ", distance: " +
8.                 String.format("%.02f", (float)car.getDistance() / 1000) + "km" +
9.                 ", time: " +
10.                 String.format("%.02f", (car.getSimulationStep() * 0.1) / 60) + "min" + " " +
11.                 ", steps: " + car.getSimulationStep() + "\n");
12.             carIndex++;
13.         }
14.         //SUMA TUTUROR TIMPILOR SI FAC MEDIA
15.
16.         int time = 0;
17.         int distance = 0;
18.         for(Vehicle vehicle : cars){
19.             time+=vehicle.getTime();
20.             distance+=vehicle.getDistance();
21.         }
22.         averageSpeed = (float)distance/time;
23.         System.out.println("Viteza medie: " + distance/time);
24.         myWriter.close();
25.         System.out.println("Successfully wrote to the file.");
26.     } catch (IOException e) {
27.         System.out.println("An error occurred.");
28.         e.printStackTrace();
29.     }
30. }
31.

```

4. Metodă ce deschide fișierul Results.txt

```

1. public static void openFile(){
2.     try
3.     {
4.         File file = new File("Results.txt");
5.         if(!Desktop.isDesktopSupported())
6.         {
7.             System.out.println("not supported");
8.             return;
9.         }
10.        Desktop desktop = Desktop.getDesktop();
11.        if(file.exists())
12.            desktop.open(file);
13.    }
14.    catch(Exception e)
15.    {
16.        e.printStackTrace();
17.    }
18. }
19.

```

5. Metodă ce numără N secunde:

```

1.     public boolean run(double time) {
2.         if (mils == ONE_SECOND * time) {
3.             mils = 0;
4.             return true;
5.         } else {
6.             mils++;
7.             return false;
8.         }
9.     }
10.

```

6. Metoda ce modifica strada pentru o mașină

```

1.     private void changeRoad(Vehicle v, Road r) {
2.         v.setRoad(r);
3.         v.setDirection(Vehicle.generateRandomDirection(r));
4.         v.setDistance(v.getRoad().getDistance());
5.     }
6.

```

7. Metodă ce verifică dacă se află o masină în fața altei mașini

```

1.     private boolean carCollision(float x, float y, Vehicle v) {
2.         for (Iterator<Vehicle> iterator = cars.iterator(); iterator.hasNext(); )
3.         {
4.             try {
5.                 Vehicle vehicle = iterator.next();
6.                 if (v.getRoad() == vehicle.getRoad()) {
7.                     if ((int) y == (int) vehicle.getY() || (int) y == (int)
vehicle.getY() - 1 || (int) y == (int) vehicle.getY() + 1) {
8.                         if (!vehicle.equals(v)) { //trebuie sa verific alte
mașini, nu masina curenta
9.                             //x = partea stanga a mașini
10.                            //x + width = partea dreapta a mașini
11.                            //u = celelalte mașini
12.                            if (((int)x < (int)vehicle.getX() +
vehicle.getWidth() - 1 && (int)x + v.getWidth() + 1 > (int)vehicle.getX()) ||
13.                                ((int)x < (int)vehicle.getX() +
vehicle.getWidth() + 1 && (int)x + v.getWidth() - 1 > (int)vehicle.getX())) {
14.                                Traffic.running = false;
15.                                Traffic.ok = false;
16.                                JOptionPane.showMessageDialog(null,
"BLOCK!!! \n" + "(on road: " +
v.getRoad().getName() + ")",
17.                                    "BLOCK", JOptionPane.ERROR_MESSAGE);
18.                            }
19.                            if (((int)x < (int)vehicle.getX() +
vehicle.getWidth() - 4 && (int)x + v.getWidth() + 4 > (int)vehicle.getX()) ||
20.                                ((int)x < (int)vehicle.getX() +
vehicle.getWidth() + 4 && (int)x + v.getWidth() - 4 > (int)vehicle.getX())) {
21.                                return false;
22.                            }
23.                        }
24.                    }
25.                    if ((int) x == (int) vehicle.getX() || (int) x == (int)
vehicle.getX() - 1 || (int) x == (int) vehicle.getX() + 1) {
26.                        if (!vehicle.equals(v)) { //trebuie sa verific alte
mașini, nu masina curenta
27.                            //y = partea de sus a mașini
28.                            //x + height = partea de jos a mașini
29.                            //u = celelalte mașini
30.

```

```

31.         if (((int)y < (int)vehicle.getY() +
vehicle.getHeight() - 1 && (int)y + v.getHeight() + 1 > (int)vehicle.getY()) ||
32.             ((int)y < (int)vehicle.getY() +
vehicle.getHeight() + 1 && (int)y + v.getHeight() - 1 > (int)vehicle.getY())) {
33.             Traffic.running = false;
34.             Traffic.ok = false;
35.             JOptionPane.showMessageDialog(null,
36.                 "BLOCK!!! \n" + "(on road: " +
v.getRoad().getName() + ")",
37.                 "BLOCK", JOptionPane.ERROR_MESSAGE);
38.             }
39.         if (((int)y < (int)vehicle.getY() +
vehicle.getHeight() - 4 && (int)y + v.getHeight() + 4 > (int) vehicle.getY()) ||
40.             ((int)y < (int)vehicle.getY() +
vehicle.getHeight() + 4 && (int)y + v.getHeight() - 4 > (int)vehicle.getY())) {
41.             return false;
42.         }
43.     }
44. }
45. }
46. } catch (ConcurrentModificationException e) {
47.     System.out.println("O puscata la threaduri");
48.     return true;
49. }
50. }
51. return true;
52. }
53.

```