System Kraksim

- dokumentacja programisty -

Kwiecień, 2016

Akademia Górniczo-Hutnicza
Wydział Infomatyki, Elektroniki i Telekomunikacji

Tytuł:
Dokumentacja programisty systemu Kraksim

Lata tworzenia aplikacji:
2006-2016

Autorzy:
Matyjewicz, Rybacki, Adamski, Pierzchała, Dziewoński, Zalewski, Kot, Skałka, Doliński, Skowron, Legień, Bibro, Bober, Liput, Haręza, Ostaszewski, Dygoń, Kawula, Ćwik, Kruczek, Grabiański, Królikowski

Opiekunowie projektu:
dr hab. inż. Jarosław Koźlak
dr inż. Małgorzata Żabińska-Rakoczy

Ilość kopii: 1

Liczba stron: 50

Data publikacji:
May 24, 2016

Opis zawartości:

- Importowanie i konfiguracja projektu
- Architektrura systemu
- Opis wykorzystywanych algorytmów
- Opis formatur plików wejściowych i wyjściowych

Spis treści

Rozdział 1

Lista pojęć i definicji

Referencja pojęć występujących w dokumentacji.

mapa Graficzna reprezentacja modelu miasta.

model ruchu Model sieci drogowej. Składa się z bram (węzły generujące ruch - samochody wychodzą z nich i wchodzą), skrzyżowań (węzłów gdzie spotykają się krawędzie grafu) oraz dróg (połączeń między węzłami - krawędzie grafu)

algorytm sotl Autonomiczna metoda sterowania światłami. Ocena dokonywana jest na podstawie sytuacji lokalnej. Światła nie komunikują się ze sobą a zmiana fazy zachodzi przy spełnieniu wyspecyfikowanych warunków zewnętrznych (np. iloczyn liczby samochodów czekających na czerwonym i czas od zapalenia czerwonego światła jest większe od ustalonego minimum)

algorytm statyczny zachowania świateł Koordynacja zmian świateł na skrzyżowaniach oparta na ustalonych stałych czasach niezależnych od stanu ruchu. Często ustala się różne czasy w zależności od pory dnia by jak najlepiej zaadaptować się do natężenia ruchu.

algorytm RL Algorytm sterowania światłami oparty o metody machine learningu.

metryka Sposób mierzenia odległości pomiędzy węzłami w modelu (np. metryka euklidesowa, taksówkarska)

algorytm głosowania Sposób wyboru głównych wezłów na mapie w oparciu o głosowanie. W panelu konfiguracyjnym wybieramy czy chcemy użyć tego algorytmu, rodzaj metryki oraz liczbę węzłów na które każdy węzeł może oddać głos.

węzły Skrzyżowania - przecięcia dróg oraz bramy - generujące ruch.

Rozdział 2

Wprowadzenie

W tym rozdziale wymienimy wymagania sytemu oraz opiszemy jak zaimportować projekt do środowiska programistycznego oraz jak uruchomić projekt.

2.1 Wspierane środowiska programistyczne

Podane poniżej środowiska nie powinny mieć większych problemów z uruchomieniem aplikacji. Inne środowiska mogą mieć możliwość importu projektu.

- IntelliJ Idea
- Eclipse

Aplikację można rozwijać także z użyciem samego środowiska Maven jednak jest to niezalecane.

2.2 Importowanie projektu

Instrukcje można znaleźć pod:

2.2.1 IntelliJ IDEA

`https://www.jetbrains.com/help/idea/2016.1/creating-and-managing-projects.html?origin=old_help`

2.2.2 Eclipse

`http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.user%2Ftasks%2Ftasks-importproject.html`

2.3 JDK

Aplikacja Kraksim wymaga do prawidłowego działania Java Development Kit w wersji 8.

2.4 Maven

Aplikacja została zrealizowana w oparciu o framework Maven. Ze względu na to należy pamiętać o wymogu stabilnego połączenia internetowego, które może być potrzeba do pobrania niezbędnych bibliotek.

2.4.1 Zależności:

- junit 4.11
- log4j 1.2.13
- testng 6.1.1
- poi 3.8
- fest-assert 1.4
- mockito-all 1.9.0
- commons-io 2.4
- weka-stable 3.6.6
- jxl 2.6.12
- java-getopt 1.0.13
- jung-jai 2.0.1
- jung-api 2.0.1
- jung-visualization 2.0.1
- jung-io 2.0.1
- jung-3d 2.0.1
- jung-graph-impl 2.0.1
- wstx-asl 3.2.6
- stax-api 1.0.1
- vecmath 1.3.1

- j3d-core 1.3.1
- java-getopt 1.0.13
- concurrent 1.3.4
- colt 1.2.0
- jfreechart 1.0.0
- commons-lang3 3.3.1
- guava 16.0.1
- commons-cli 1.2

2.5 Uruchamianie aplikacji

Aplikację można uruchomić za pomocą metody *main(String[] args)* w klasie *pl.edu.agh.cs.kraksim.KraksimRunner*

Rozdział 3

Struktura systemu

W tym rozdziale objaśnimy ogólną architekturę systemu z wyszczególnieniem jego poszeczególnych elementów oraz ich funckcjonlności.

3.1 Budowa systemu

3.1.1 Ogólna architektura systemu

W systemie KRAKSIM równocześnie współpracuje wiele podsystemów. Aby umożliwić łatwą podmianę implementacji poszczególnych podsystemów, Kraksim ma **budowę modułową**.

Całą aplikację można więc podzielić na *jądro* - które składa się podstawowowych elementów modelu miasta budujących topologię sieci dróg oraz *moduły*, które mają określone zadania, a ich budowanie opiera się na implementacji zdefiniowanych dla nich interfejsów.

Wszystkie moduły rozszerzające potrzebne do zbudowania symulatora umieszczone zostały na rysunku poniżej. Konkretne implementacje modułów oraz definicje ich interfejsów zostały przedstawione w dalszych rozdziałach dokumentacji.

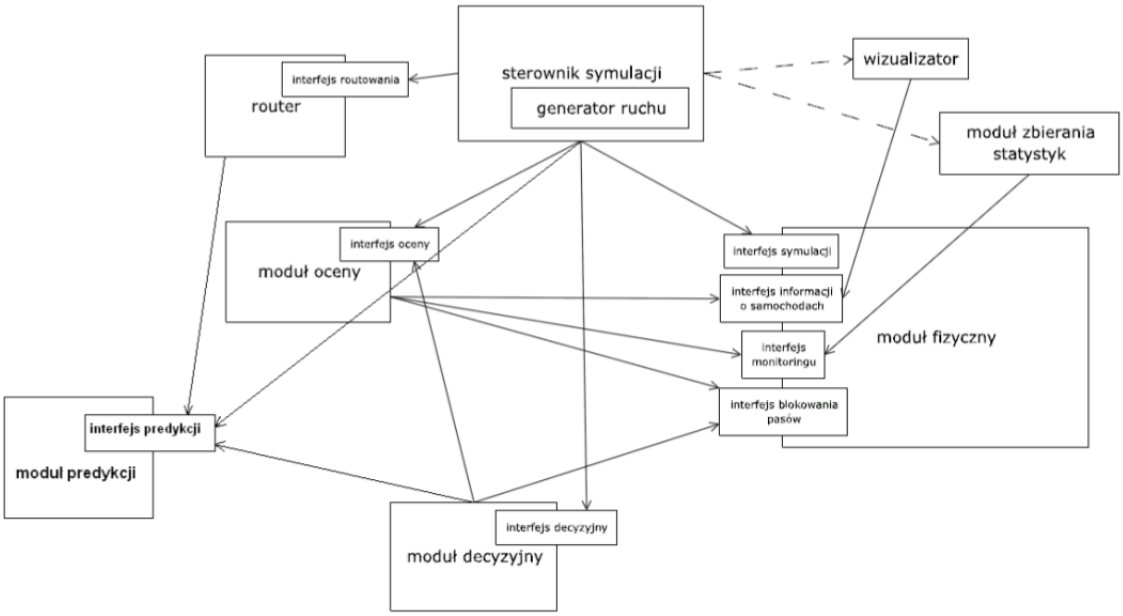

Figure 3.1: Przedstawienie modułowej architektury systemu. Dla czytelności nie zamieszczono jądra, z którego korzysta każda składowa potrzebująca informacji o topologii miasta.

Podstawowe moduły

- Moduł fizyczny

  zajmuje się symulacją ruchu, udostępnianiem informacji o tej symulacji oraz umożliwia wpływ na nią poprzez sterowanie blokadą pasów

- Moduł oceny

  zajmuje się oceną korzyści z włączenia zielonego światła

- Moduł decyzyjny

  zajmuje się podejmowaniem i realizacją decyzji o zmianie świateł

- Moduł predykcji

  poprawia moduł oceny o metody predykcji zagęszczeń na drogach i tworzenia korków, tak by wydajniej sterować sygnalizacją

- Moduł wyznaczania tras

  zajmuje się wyznaczaniem tras przejazdów dla poszczególnych pojazdów tak by uniknąć miejsc o nadmiernym zatłoczeniu

- Moduł zbierania statystyk

  służy do zbierania statystyk z symulacji, aby mogły być później udostępnione użytkownikowi

- Wizualizator

  służy do graficznej wizualizacji symulacji

- Sterownik systemu

  steruje przebiegiem symulacji

- Generator ruchu

  wydzielona część sterownika systemu służąca generowaniu kierowców oraz nadzorowaniu rozpoczynania ich podróży

3.1.2 Jądro

Centralnym składnikiem modułowej koncepcji systemu Kraksim jest *jądro*. Elementy wchodzące w skład jądra tworzą *topologię sieci* dróg, a jego jedyną funkcjonalnością jest odkrywanie tej topologii.

Elementy jądra jak np. pas ruchu można rozszerzać dodając nowe dane i operacje. Zestaw rozszerzeń elementów realizujących konkretną funkcjonalność nazywamy modułem. Interfejsem modułu nazywamy sumę wszystkich interfejsów rozszerzeń wchodzących w skład tego modułu.

Model miasta

Klasy służące reprezentacji modelu dróg znajdują się w pakiecie *pl.edu.agh.cs.kraksim.core.visitors*.

Należą do nich:

- **Element** - klasa bazowa dla wszystkich elementów modelu miasta. Zdefiniowane są w niej mechanizmy umożliwiające rozszerzenie aplikacji.

- **City** - klasa reprezentująca całe miasto. Agreguje w sobie obiekty typu Gateway, Intersection oraz Link.

- **Gateway** - klasa dziedzicząca po Node. Reprezentuje węzły wlotowe w mieście.

- **Intersection** - klasa dziedzicząca po Node. Stanowi abstrakcję dla skrzyżowań. Agreguje w sobie połączenia wchodzące i wychodzące do węzła.
- **Node** - abstrakcyjna klasa bazowa dla klas Gateway i Intersection. Deklaruje abstrakcyjne metody obsługi połączeń wchodzących i wychodzących oraz sygnalizacji, które konkretna klasa algorytmu musi implementować.

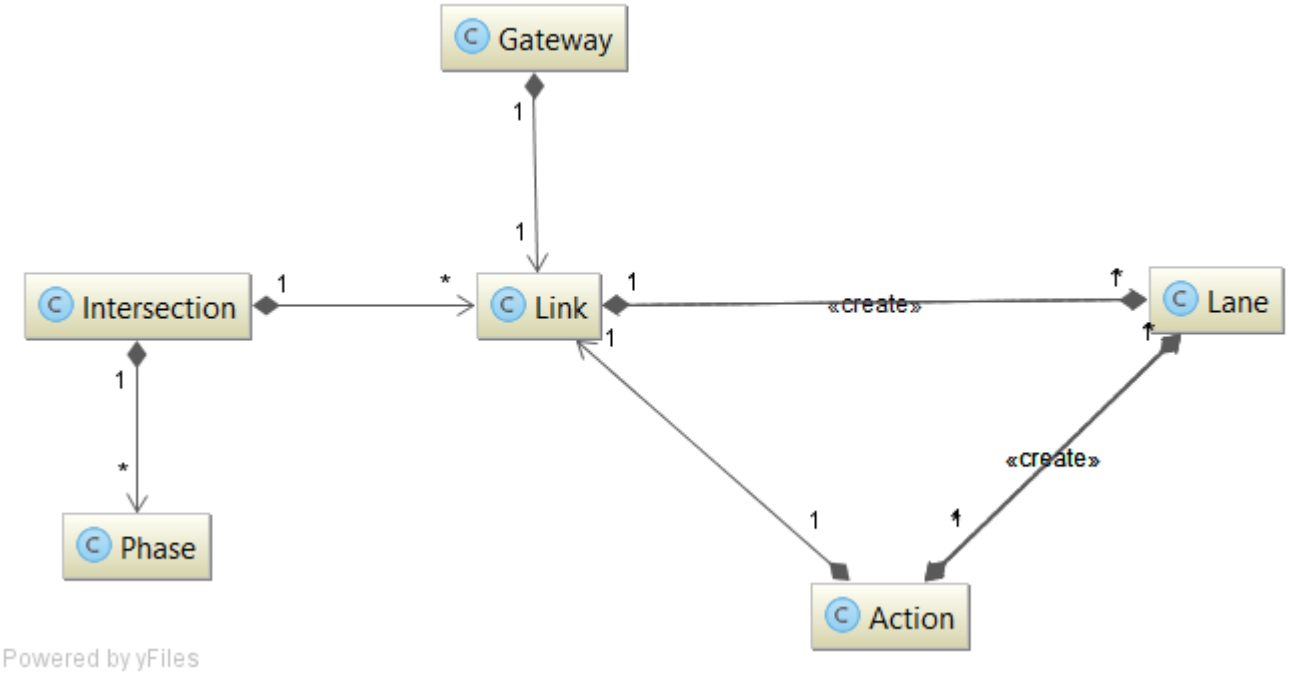

Figure 3.2: Diagram zależności pomiędzy najważniejszymi klasami modelu miasta

- **Link** - klasa implementująca połączenie pomiędzy dwoma węzłami. Zawiera metody potrzebne do skonstruowania drogi oraz iterator dróg, do których można dojechać pasami tej drogi. Link agreguje obiekty typu Lane.
- **Lane** - klasa reprezentująca pas drogowy. Agreguje akcje związane z pasem, które samochody mogą z niego wykonać. Umożliwia znalezienie dostępnych akcji, iterację po nich, w jej obrębie wprowadzone jest ograniczenie prędkości.
- **Action** - klasa związana bezpośrednio ze skrzyżowaniami. Ma ona źródło w pasie Lane, a jej celem jest Link (w obrębie którego wybierany jest główny Lane). Zawiera także informacje, które pasy mają pierwszeństwo.
- **Phase** - klasa reprezentująca fazę sygnalizacji świetlnej. Posiada atrybuty listy stanów, nazwy, czasu trwania, kierunku synchronizacji. Lista faz przechowywana jest przez obiekt typu Intersection tworząc plan sygnalizacji świetlnej dla skrzyżowania.

3.1.3 Interfejsy modułów

Poniżej opisujemy interfejsy wymienionych wcześniej modułów, które rozszerzają funkcjonalność elementów z modelu miasta.

Widoki interfejsów

Do każdego interfejsu modułu dołączona jest klasa widoku dziedzicząca po **ModuleView**.

Podklasy ModuleView pełnią dwie role:

- podają informację co dane rozszerzenie interfejsu oferuje
- umożliwia zobaczenie szczegółów modułu poprzez jego rozszerzenie

Klasa ta definiuje metody dostępu do poszczególnych modułów w postaci **ext(NAZWAMODUŁU)**

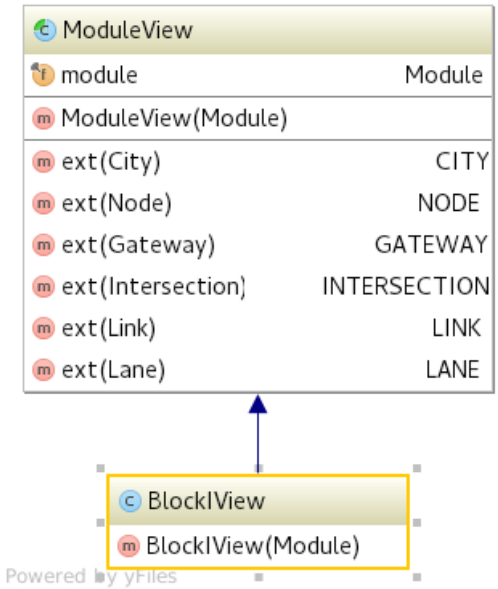

Figure 3.3: Diagram klasy ModuleView

Symulacja ruchów samochodów

Interfejs symulacji ruchów samochodów **SimIView** jest zdefiniowany w pakiecie *pl.edu.agh.cs.kraksim.sim*. Służy on przemieszczaniu samochodów tura po turze po połączeniach oraz umieszczaniu samochodow w symulacji.

Interfejs składa się z widoku **SimIView**, który umożliwia pobranie interfejsów miasta **CitySimIface** oraz przejścia **GatewaySimIface**.

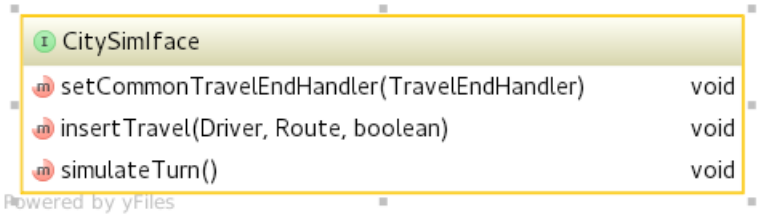

Figure 3.4: Diagram klasy CitySimIface

CitySimIFace - interfejs rozszerzenia elementu jądra **City**. Zawiera w sobie syngatury trzech metod:

- *void setCommonTravelEndHandler(TravelEndHandler handler);*

  ustawia handler obsługujący zdarzenie zakończenia podróży, wspólny dla wszystkich węzłów wlotowych

- *void insertTravel(Driver driver, Route route, boolean rerouting);*

  umieszcza samochód z trasą w symulacji

- *void simulateTurn();*

  wykonuje turę symulacji

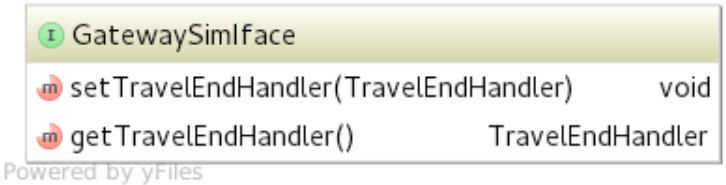

Figure 3.5: Diagram klasy GatewaySimIface

GatewaySimIface - interfejs rozszerzenia elementu jądra **Gateway**. Definiuje sygnatury dwóch metod:

- *setTravelEndHandler(TravelEndHandler handler)*

  ustawia handler obslugujący zakończenie podróży w danym węźle

- *TravelEndHandler getTravelEndHandler()*

  getter zwracający handler obsługujący zakończenie podróży w danym węźle

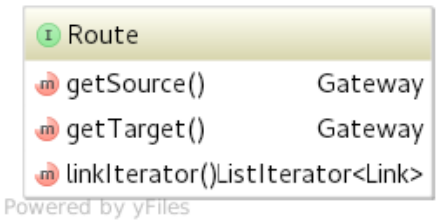

Figure 3.6: Diagram klasy **Route**

Interfejs **Route** używany przez metodę *insertTravel()* w **CitySimIface** zawiera sygnatury metod pobrania informacji o źródle i celu podróży oraz iterator kolejnych dróg trasy, który powinien generować trasę dynamicznie, w trakcie symulacji:

- *Gateway getSource();*
- *Gateway getTarget();*
- *ListIterator<Link> linkIterator();*

Interfejs informacji o samochodach

Interfejst zdefionwany jest w pakiecie *pl.edu.agh.cs.kraksim.iface.carinfo*. Służy on do uzyskiwania informacji na temat samochodów uczestniczących w ruchu.

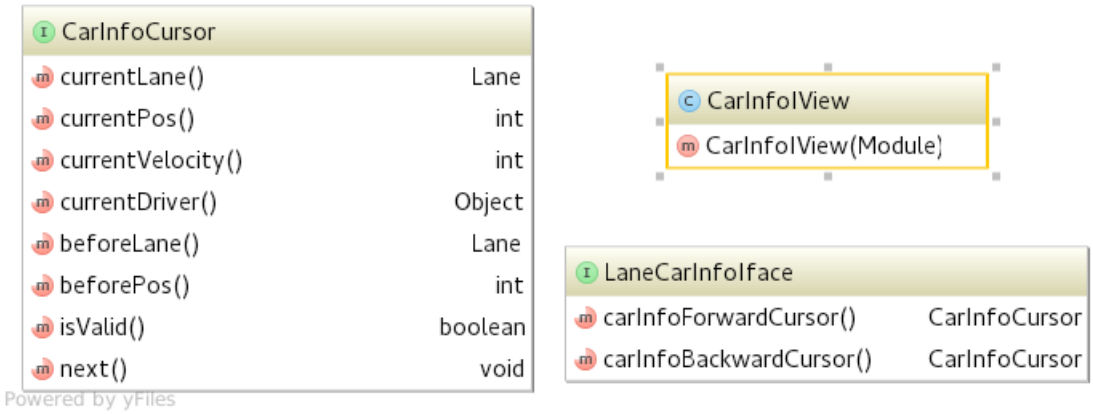

Figure 3.7: Diagramy interfejsów i klas dla pakietu carinfo

Składa się z widoku **CarInfoIView**, który służy pobraniu jedynego tutaj interfejsu rozszerzeń **LaneCarInfoIface**.

LaneCarInfoIface to interfejs rozszerzający element jądra **Lane**. Implementacja **LaneCarInfoIface** powinna definiować dwie metody:

- *CarInfoCursor carInfoForwardCursor();*
  umożliwia pozyskanie kursora wskazującego na pojazdy zaczynając od początku pasa
- *CarInfoCursor carInfoBackwardCursor();*
  umożliwia pozyskanie kursora wskazującego na pojazdy zaczynając od końca pasa

Obie metody zwracają obiekt typu **CarInfoCursor**, który definiowany jest przez interfejs zawierający metody:

- *Lane currentLane()*
  zwraca Lane na którym samochód zakończył ostatni ruch
- *int currentPos()*
  zwraca aktualną pozycję pojazdu na pasie
- *int currentVelocity()*
  zwraca ilość komórek, jaką przebył pojazd w ostatnim ruchu
- *Object currentDriver()*
  zwraca kierowcę samochodu
- *Lane beforeLane()*
  zwraca pas, z którego samochód zaczynał ostatni ruch
- *int beforePos()*
  zwraca pozycję pasa, z której samochód rozpoczynał ostatni ruch
- *boolean isValid()*
  zwraca czy kursor wskazuje na jakiś samochód
- void next()
  przesuwa kursor do następnego samochodu

Interfejs monitoringu

Interfejs zdefiniowany w pakiecie *pl.agh.edu.cs.kraksim.iface.mon*. Interfejs ten umożliwia rejestrację akcji, która zostanie wykonana po przejściu samochód przez określony punkt połączenia, zniknięcie lub pojawienie się na mapie.

Interfejs tak jak poprzednie posiada klasę widoku **MonIView**, z której mozna wyciągnąć 3 interfejsy rozszerzające klasy jądra: **Link**, **Lane** oraz **Gateway**.

GatewayMonIface - interfejs rozszerzenia elementu **Gateway**. Definiuje następujące sygnatury:

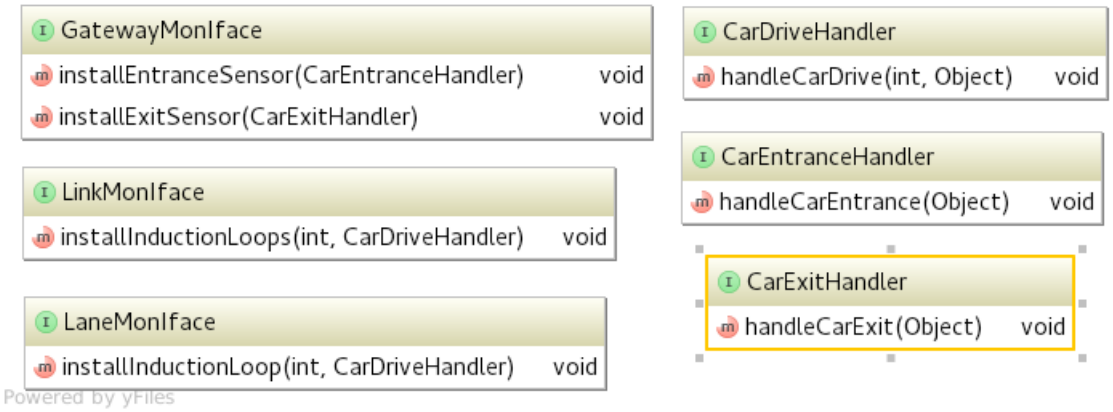

Figure 3.8: Diagramy interfejsów i klas dla pakietu mon

- *void installEntranceSensor(CarEntranceHandler handler)*

  rejestracja handlera obsługującego pojawienie się pojazdu

- *void installExitSensor(CarExitHandler handler)*

  rejestracja handlera obsługującego zniknięcie pojazdu

LaneMonIface - interfejs rozszerzenia klasy **Lane**. Definiuje jedną sygnaturę:

- *installInductionLoop(int line, CarDriveHandler handler)*

  rejestracja handlera obsługującego zdarzenie przejazdu przez samochód określonego fragmentu pasa (dokładnie linii między komórkami o numerach *line-1* oraz *line*

LinkMonIface - interfejs rozszerzenia klasy **Link** z metodą podobną jak powyższy:

- *installInductionLoops(int line, CarDriverHandler handler)*

  rejestracja handlera obsługującego zdarzenie przejazdu przez samochód określonego fragmentu połączenia

Powyższe metody interfejsu modułu monitoringu używają jako swoich argumentów handlerów, które zostały również zdefiniowane w tym interfejsie.

- **CarEntranceHandler** - *handleCarEntrance()*
- **CarExitHandler** - *handleCarExit()*
- **CarDriveHandler** - *handleCarDrive()*

  Wszystkie definiują po jednej metodzie obsługi konkretnych zdarzeń.

Interfejs blokowania pasów

Interfejst zdefiniowany w pakiecie *pl.edu.agh.cs.kraksim.iface.block*.

Interfejs ten pozwala na blokowanie pasa ruchu. Blokowanie powoduje, że samochód znajdujący się na danym pasie nie może z niego wyjechać do kończącego pas węzła. Jest to uogólnienie sterowania światłami, bowiem czynność zapalania świateł na skrzyżowaniach sprowadza się do blokowania/odblokowania pasu ruchu.

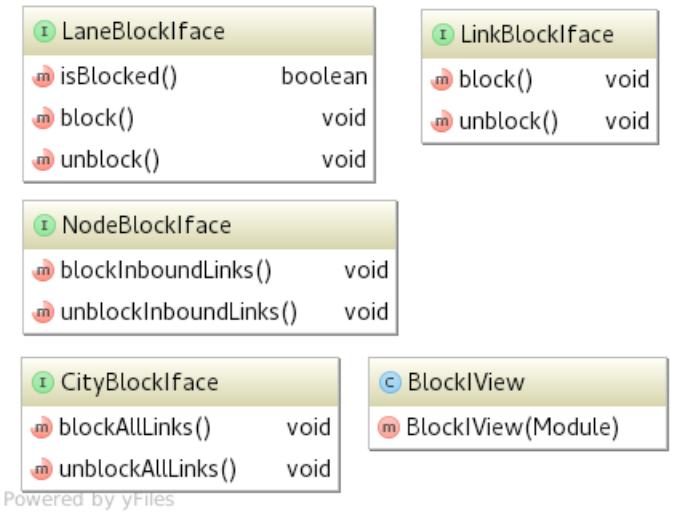

Figure 3.9: Diagramy interfejsów i klas dla pakietu block

Widokiem interfejstu jest klasa **BlockIView**. Rozszerzenie to dotyczy 4 elementów miasta: **CityBlockIface** - rozszerzenie elementu **City**. Definiuje dwie sygnatury:

- *blockAllLinks()*

  blokada wszystkich pasów w całym mieście

- *unblockAllLinks()*

  odblokowanie wszystkich pasów w całym mieście

NodeBlockIface - interfejs rozszerzania elementu **Node**.

- *blockInboundLinks()*

  blokuje wszystkie połączenia wchodzące do węzła

- *unblockInboundLinks()*

  odblokowuje wszystkie połączenia wchodzące do węzła

LinkBlockIface - interfejs rozszerzenia elementu **Link**.

- *block()*

  blokuje wszystkie pasy na danym połączeniu (**Link**)

- *unblock()*

  odblokowuje wszystkieg pasy na danym połączeniu (**Link**)

LaneBlockIface - interfejs rozszerzenia elementu **Lane**.

- *block()*

  blokuje pas ruchu

- *unblock()*

  odblokowuje pas ruchu

Interfejs oceny korzyści z zielonego światła

Interfejs zdefioniowany w pakiecie *pl.agh.edu.cs.kraksim.iface.eval*.

Interfejs odnosi się do optymalizatora pracy świateł. Służy do sumarycznej oceny korzyści dla wszystkich samochodów z danego pasa, gdy będzie na nim świeciło się zielone światło. Polityka oceny powinna być spójna w ramach jednego skrzyżowania.

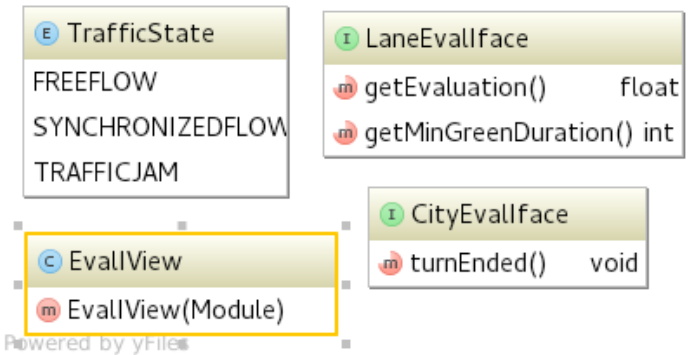

Figure 3.10: Diagramy interfejsów i klas dla pakietu *eval*

Rozszerzenie to obejmuje dwa elementy miasta **Lane** oraz **City**.
EvalIView - klasa będąca widokiem interfejsów wchodzących w skład rozszerzenia.
CityEvalIface - rozszerzenie elementu **City**.

- void *turnEnded()*

  uruchamia proces obliczania kosztów, służy jedynie do informawania modułu oceny, że symulacja tury zakończyła się

LaneEvalIface - rozszerzenie elementu **Lane**.

- *float getEvaluation()*

  ocena sytuacji drogowej na danym pasie, oceny pasów wchodzących w skład skrzyżowania są porównywane i na tej podstawie można obliczyć koszt zmiany stanu sygnalizacji

- *int getMinGreenDuration()*

  zwraca minimalny czas trwania następnej fazy zielonego światła (podawane w liczbie tur)

Interfejs decyzyjny

Interfejs zdefiniowany w pakiecie *pl.edu.agh.cs.kraksim.iface.decision*.

Interfejs służy do informowania modułu realizującego sterowanie światłami o końcu tury (symulacji ruchu) i pozwala na wykonanie niezbędnych działań.

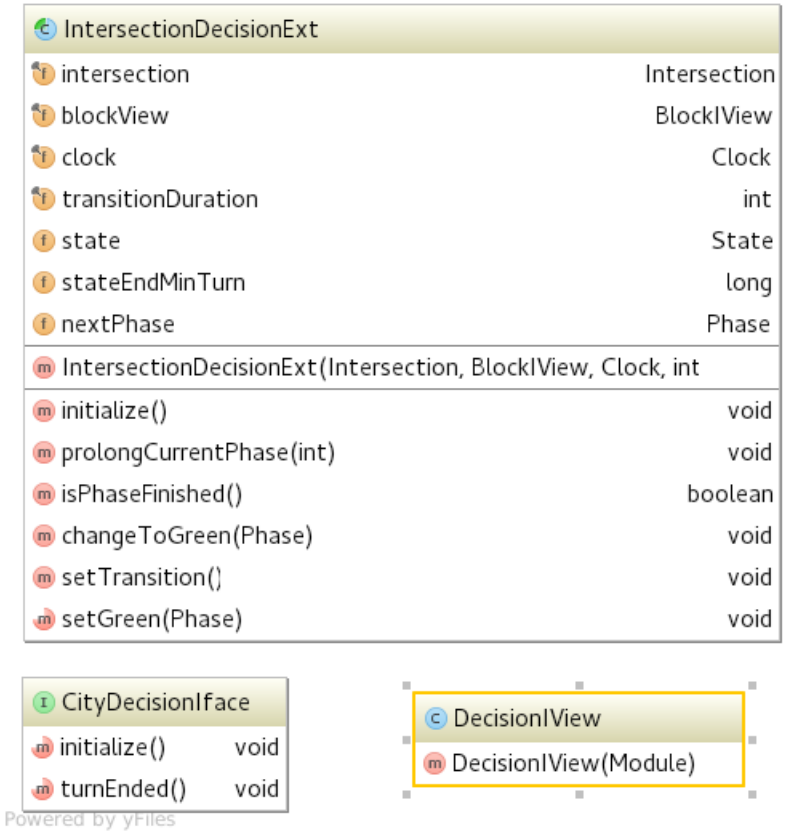

Figure 3.11: Diagramy interfejsów i klas dla pakietu *decision*

Intefejs składa się z widoku **DecisionIView**, który jest widokiem udostępniającym jeden interfejs rozszerzający.

CityDecisionIface - interfejs rozszerzenia elementu **City**. Definiuje dwie sygnatury:

- *void initialize()*

  metoda wywoływana podczas inicjalizacji systemu

- *void turnEnded()*

  metoda wywoływana po każdej zakończonej turze symulacji

3.2 Podstawowe elementy systemu

3.3 Implementacja modułu fizycznego

Moduł fizyczny jak już opisano w sekcji ogólnej architektury systemu ma na celu wykonywanie symulacji, udostępnianie informacji o niej oraz pozwala na blokowanie pasów w topologii miasta.

Implementacja modułu fizycznego znajduje się w pakiecie *pl.edu.agh.cs.kraksim.real*. Działanie modułu symulacyjnego oparte jest na algorytmie ***Nagla-Schreckenberga***.

3.3.1 Mikroskopowy model symulacji ruchu w Kraksim

Zaletą modelu symulacji Nagla-Schreckenberga jest prostota implementacji i szybkość działania nawet dla dużych i skomplikowanych sieci drogowych.

Model ten zakłada, że pas ruchu składa się z jednowymiarowej tablicy komórek, z której każda może być zajęta maksymalnie przez jeden pojazd. Pojazdy poruszają się wzdłuż tablicy zajmując kolejne pola. Ich prędkość wyrażona jest jako liczba komórek na turę symulacji.

Turę symulacji można podzielić na 5 etapów:

1. **Zmiana pasa** - następuje próba zmiany pasa w celu wykonania manewru wyprzedzania lub w celu ustawienia się na wybranym pasie, z którego chce się wyjechać na skrzyżowanie.
2. **Przyspieszanie** - jeśli samochód n nie jedzie z maksymalną szybkością i odległość od poprzedzającego pojazdu umożliwia zwiększenie prędkości tak by nie było kolizji, prędkość wzrasta o 1 komórkę na turę.
3. **Zwalnianie** - jeśli samochód n znajduje się w odległości od poprzedzającego pojazdu mniejszej niż ta jaką samochód miałby pokonać w następnej turze, zmniejsza się prędkość pojazdu o 1.
4. **Czynnik losowy** - prędkość pojazdu zmniejszana o 1 z pewnym prawdopodobieństwem p
5. **Ruch** - samochody przemieszczane są w czasie tury o liczbę komórek zgodną ze swoją wartością szybkości v.

3.3.2 Implementacja modelu ruchu

Klasa implementująca algorytm Nagla-Schreckenberga to **LaneRealExt** - rozszerzająca **Lane**, implementuje ona 3 interfejsy: **LaneBlockIface**, **LaneCarInforIface**, **LaneMonIface**.

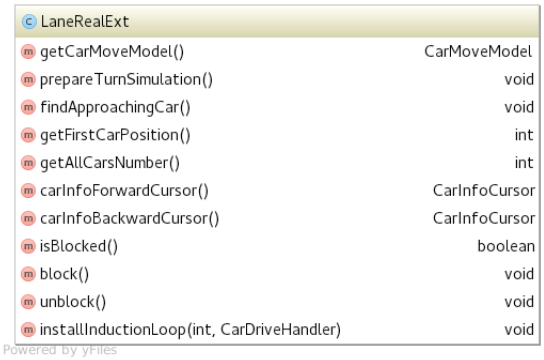

Figure 3.12: Diagram klasy LaneRealExt implementującej algorytm Nagla-Schreckenberga

3.4 Algorytmy sterowania sygnalizacją świetlną

W systemie zaimplementowane zostały 3 rodzaje sterowników sygnalizacji świetlnej:

- sterownik z fazami o stałej długości
- SOTL
- RL

W poniższych sekcjach przedstawiamy szczegóły algorytmu oraz sposób ich realizacji w ramach **modułu decyzyjnego**.

3.4.1 Przegląd algorytmów

Stałe fazy

Najprostszy sterownik sygnalizacji świetlnej. Działa w oparciu o stałą konfigurację przygotowaną przez użytkownika. Do jego działania wystarczą: zegar i dostarczona konfiguracja planu fazowego.

Konfiguracja planu fazowego to sekwencja faz wraz z opisem, każdej z nich. Opis fazy to kolory wszystkich świateł na danym skrzyżowaniu i czas trwania tej fazy.

Szczegóły implementacyjne

3.4.2 SOTL

SOTL - Self-Organizing Traffic Lights (samoorganizująca się sygnalizacja świetlna). Metoda, w której kontroler sygnalizacji utrzymuje informację o liczbie pojazdów c na pasie, na którym w tym momencie pali się czerwone światło oraz o czasie t jaki upłynął od zapalenia czerwonego światła. Gdy iloczyn $c * t$ przekroczy ustaloną wartość Φ następuje wymuszenie zmiany światła na zielone na tym pasie co pociąga ze sobą zmianę fazy pracy świateł dla całego skrzyżowania.

Aby zapobiec zbyt częstym zmianom świateł wprowadza się minimalny czas ϕ_{min}, który musi zostać przekroczony by nastąpiła zmiana.

Dodatkowo by ograniczyć zaburzanie płynności ruchu przez algorytm dodaje się jeszcze dwa warunki: **1)** czerwone światło nie jest zmieniane na zielone jeśli na prostopadłej ulicy przynajmniej jeden pojazd zbliża się do skrzyżowania i jest w odległości mniejszej niż ω od niego oraz **2)** jeśli do skrzyżowania w prostopadłym kierunku zbliża się jednak więcej niż *mi* samochodów, wówczas nie bierze się pod uwagę pierwszego ograniczenia.

Szczegóły implementacyjne W projekcie zaimplementowany SOTL używa metody SOTL-phase. Implementacja dotyczy modułu decyzyjnego.

SOTL do poprawnego działania wymaga zegara, dlatego korzysta z interfejsu **Clock**, który dostarczany jest przez moduł symulacji. Aby umożliwić zliczanie samochodów na poszczególnych pasach skrzyżowania zaimplementowano rozszerzenie w interfejsie oceny korzyści z zielonego światła: **LaneSOTLExt** - implementujące interfejs **LaneEvalIface** oraz **CitySOTLExt** - implementujące interfejs **CityEvalIface**.

Ocena liczby pojazdów w liniach, która jest potrzebna w tej metodzie oceny korzyści opiera się na rozszerzeniu interfejsu monitoringu. W metodzie SOTL instaluje się tu dwie pętle indukcyjne - jedną na początku pasa i drugą na końcu. Liczba samochodów na pasie liczona jest kontrolując samochody przybywające (w jednej pętli) i wyjeżdżające (w drugiej).

LaneSOTLExt implementuje metody interfejsu w natępujący sposób:

```
void turnEnded() {
        if (laneBlockExt.isBlocked()) {
                sotlLaneValue += carCount;
        } else {
                sotlLaneValue = 0;
        }
}

public float getEvaluation() {
        LOGGER.trace(id + " carCount=" + carCount + ",
                sotlValue=" + sotlLaneValue + ", blocked=" +
                laneBlockExt.isBlocked());
        if (sotlLaneValue > params.threshold) {
                return sotlLaneValue;
        } else {
                return 0;
        }
}

public int getMinGreenDuration() {
        int ret = (int) ((carCount) * (float) params.carStartDelay
        + (carCount / (float) params.carMaxVelocity));
```

```
        return Math.max(ret, SOTLParams.minimumGreen);
}
```

CitySOTLExt natomiast następująco:

```
public void turnEnded() {
        for (Iterator<Link> linkIter = city.linkIterator();
    linkIter.hasNext(); ) {
                Link link = linkIter.next();
                for (Iterator<Lane> laneIter = link.laneIterator();
        laneIter.hasNext(); ) {
                        Lane lane = laneIter.next();
                        ev.ext(lane).turnEnded();
                }
        }
}
```

Metoda *turnEnded()* informuje, że symulacja tury zakończyła się i można przystąpić do obliczania oceny. I tak tutaj sprawdzane jest w warunku, czy pas ruchu jest zablokowany. Jeżeli tak to obliczana jest wartość oceny dodając liczbę samochodów znajdujących się na pasie. Jeśli jednak dla pasa świeci sie zielone światło to ocena przyjmuje wartość zero.

float getEvaluation() wywoływana jest gdy należy porównać oceny dla pasów na skrzyżowaniu. W metodzie kontrolowana jest wartośc sotlLaneValue i niezerowa ocena zwracana jest tylko wtedy gdy wartość sotlLaneValue będzie większa od predefiniowanego thresholdu.

int getMinGreenDuration() zwracać ma minimalny czas trwania zielonego światła po zmianie świateł (podawane w liczbach tur). Wartość ta liczona jest jako iloczyn liczby samochodów czekajacych i opóźnienia startu samochodu carStartDelay. Do wartości tej dodany jest iloraz liczby samochodów i maksymalnej dopuszczalnej prędkości (w komórkach na turę) - po to by ostatni samochód czekający zdążył na tym świetle przejachać. Jeżeli tak obliczona wartość nie przekracza parametru minimumGreen, wówczas zwracana jest wartość stała minimumGreen.

Metoda *turnEnded()* w klasie CitySOTLExt wywoływana jest globalnie po skończeniu tury. Iteruje ona po wszystkich połączeniach **Link**. I dla każdego połączenia wywołuje turnEnded() na wszystkich pasach **Lane** wchodzących w jej skład.

Sposób konfiguracji

Można konfigurować na etapie kompilacji następujące parametry:

- *zoneLength* - długość strefy
- *carStartDelay* - ilość czasu po jakim ruszają kolejne samochody w korku
- *carMaxVelocity* - maksymalna prędkość samochodu

- *threshold* - próg, którego przekroczenie pozwala na branie pod uwagę wartości pasa podczas oceny (nie wynosi ona 0)
- *minimumGreen* - minimalny czas trwania zielonego światła, jeśli jest tylko co najwyżej jeden pojazd w strefie

W pliku konfiguracyjnym można skonfigurować jeden parametr podając go w postaci:
algorithm = sotl:zone=18
gdzie *zone* odpowiada parametrowi *zoneLength*. Parametr *threshold* jest ustalony w kodzie na *zoneLenght-5*, parametr *minimumGreen* na 5.

3.4.3 RL

RL - reinforcement learning. Algorytm realizujący strategię uczenia ze wzmocnieniem.

Algorytm rozwiązuje problem wyboru optymalnej konfiguracji świateł w danej chwili. Konfiguracja ta może być wybrana z predefiniowanego zbioru lub tworzona dynamicznie.

Konieczne jest przyjęcie funkcji kosztu. Jeśli samochód w danej turze musi czekać funkcja kosztu wyniesie dla niego 1, a w przeciwnym wypadku 0.

W celu optymalizacji używa się dwóch funkcji szacujących ilość kary nałożonej na samochód do końca podróży:

- $Q(s, l)$ - szacuje ilośc kary nałożonej na samochód będący w stanie s, gdy znane jest światło l na skrzyżowaniu.
- $V(s)$ - szacuje ilość kary nałożonej na samochód będący w stanie s gdy nie jest znane światło na skrzyżowaniu.

Moduł oceny RL podobnie jak SOTL, implementuje tylko rozszerzenie pasa ruchu oraz rozszerzenie miasta wraz z paroma klasami pomocniczymi.

Klasy rozszerzające moduł oceny implementują interfejsy: **CityEvalIface** - **CityRLExt** oraz **LaneEvalIface** - **LaneRLExt**. Widok **ModuleView** implementuje klasa **RLEView**.

3.5 Algorytmy koordynacji sterowania sygnalizacją

Do koordynacji zmian faz sygnalizacji pomiędzy skrzyżowaniami Kraksim używa alogrytmu *OptAPO* - Optimal Asynchronous Partial Overlay. Idea tej metody polega na utworzeniu grup współpracujących agentów-skrzyżowań. Wtedy w topologii miejskiej tworzą się kanały lub arterie, dla których ustalony jest jeden wspólny, główny kierunek ruchu.

Każdy agent wykonuje podstawowe kroki algorytmu, w celu poprawienia lokalnego rozwiązania, przy czym agent może zmienić lokalne rozwiązanie tylko pod warunkiem, że poprawi ono wartośc globalnego rozwiązania.

W zaimplementowanym rozwiązaniu każdy agent ma przypisaną jedną zmienną d będącą kierunkiem synchronizacji ($d \in \{NS/SN, EW/WE\}$

Koordynacja w jednym kierunku sprawia, że agent-sterownik ustala swoje fazy świateł zgodnie z sąsiadami ulokowanymi na wybranym kierunku. Jeżeli czasy planu sygnalizacji ustalone sa tak, żeby dac priorytet kierunkowi NS/SN, to nie jest możliwa równoczesna koordynacja w prostopadłym kierunku.

Funkcja kosztu liczona jest na podstawie liczby pojazdów dojeżdżających do skrzyżowania oraz kierunku synchronizacji. Koszt dla danego agenta zależy od kierunku, w którym ruch ma największe natężenie. Agent liczy koszt jako sumę kosztów z agentami sąsiadującymi w tym konkretnym kierunku. Funkcja kosztu pomiędzy dwoma agentami obliczana jest następująco:

- kierunki synchronizacji są takie same oraz jest to synchronizacja w kierunku wyższego natężenia - koszt wynosi 0.
- kierunki synchronizacji są różne, ale agent i używa planu sygnalizacji w kierunku, w którym występuje większe natężenie - koszt wynosi: $\frac{\#pojazdwjadcychzx_j do x_i}{\#pojazdwjadcychdoskrzyowaniax_i}$
- kierunki synchronizacji są różne oraz agent i uzywa planu sygnalizacji w kierunki innym, niż ten w którym występuje największe natężenie - koszt: $2 \times \frac{\#pojazdwjadcychzx_j do x_i}{\#pojazdwjadcychdoskrzyowaniax_i}$

3.5.1 Szczegóły implementacyjne

Rozwiązanie *OptAPO* zostało zaimplementowane na poziomie **modułu decyzyjnego**. Ponieważ w systemie żadne inne rozwiązanie nie korzysta z modułu decyzyjnego wybierającego kierunki synchronizacji, nie rozdzielono modułu oceny i modułu decyzyjnego.

Pakiet implementujący rozwiązanie to *pl.edu.agh.cs.kraksim.dsyncdecision*. Jak wyżej wspomniano agreguje ona moduł oceny oparty na metodzie *OptAPO*.

Moduł podobnie jak pozostałe rozszerzenia moduły decyzyjnego składa się z implementacji interfejsu **CityDecisionIface** - **CityDSyncDecisionExt** oraz klasy **IntersectionDecisionExt**.

Każdemu rozszerzeniu skrzyżowania odpowiada jeden agent optymalizujący kierunek synchronizacji. Agenty działają na prostej platformie agentowej, implementowanej wewnętrznie w systemie, przypisanej do rozszerzenia miasta. Rozszerzenie miasta jest odpowiedzialne za inicjalizacji agentów i skojarzenie ich z odpowiednimi skrzyżowaniami.

TODO struktura implementacji

3.6 Optymalizacja ruchu SNA

Kraksim pozwala również wykorzystać optymalizację ruchu drogowego przy użyciu algorytmów bazujących na sieciach społecznych *SNA*.

3.6.1 Teoria

Pierwszą fazą algorytmu jest dokonanie podziału na podgrafy. Jako metodę podziału wykorzystuje się metodę k-means. Klasteryzacja przebiega dwuetapowo:

1. **Wybranie środków klastrów** - wierzchołków o najwyższej wartości miar
2. Dla wszystkich wierzchołków (skrzyżowań) sprawdzamy do którego środka klastra mają najbliżej. Węzeł zostanie **przypisany do tego klastra**, do którego wierzchołka głównego ma najbliżej.

Następną fazą algorytmu jest wprowadzenie hierarchii wewnątrz podgrafów.
Na bazie wartości miar węzłów wprowadza się wielopoziomową strukturę hierarchiczną o liczbie poziomów ustalonej w konfiguracji symulacji.

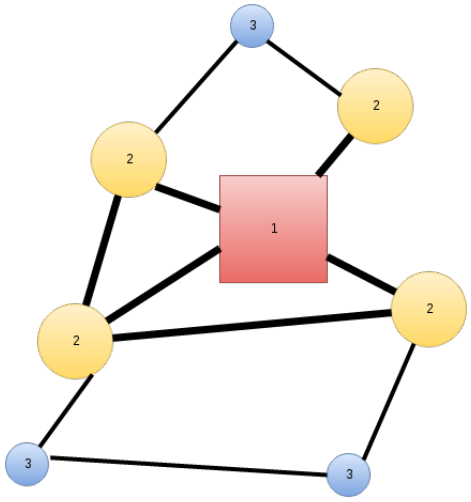

Figure 3.13: Graf reprezentujący przykładowy klaster z zaprezentowaniem kolejnych poziomów hierarchii

Hierarchia ta jest wykorzystywana do synchronizacji świateł. Światła na niższym poziomie są ustawiane na podstawie ustawień skrzyżowań znajdujących się wyżej w hierarchii. Synchronizacja propaguje się więc od w dół hierarchii od węzłów ważniejszych do mniej ważnych.
Ważnym elementem algorytmu jest możliwość synchronizacji pomiędzy podgrafami. Osiąga się to poprzez wymianę informacji pomiędzy środkami podgrafów sąsiadujących ze sobą, tak by korki między nimi były jak najmniejsze.

3.6.2 Mechanizmy synchronizacji świateł

Zaimplementowany algorytm w pierwszej kolejności konfiguruje światła przez główne skrzyżowania i następnie wymienia informację pomiędzy sąsiednimi klastrami. Następnie gdy główne skrzyżowania skorygują swoje ustawienia na podstawie wymienionych ze sobą informacji, propagują informację wewnątrz swoich klastrów.
Węzły podrzędne po otrzymaniu informacji od skrzyżowania głównego przystępują do konfiguracji własnych świateł.

3.6.3 Wizualizacja grafu

Widok reprezentacji grafu znajduje się w drugiej zakładce głównego okna symulacji. Został on stworzony przy wykorzystaniu biblioteki *JUNG*. Graf obrazuje podział miasta na podgrafy. Poszczególne węzły mają nazwy analogiczne do skrzyżowań. Rozmieszczenie węzłów w grafie naśladuje w pewnym stopniu rozmieszczenie rzeczywistych skrzyżowań na mapie.

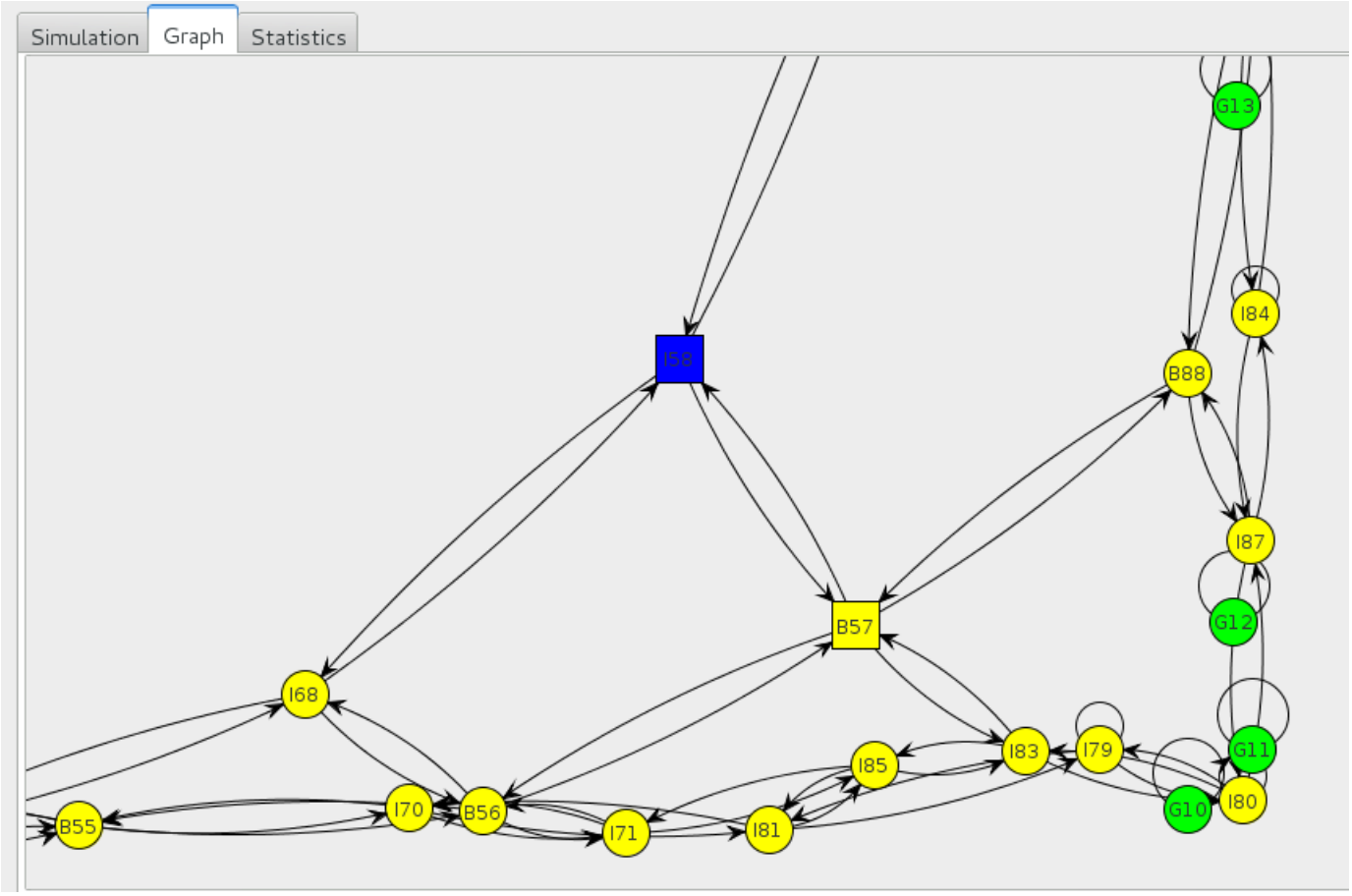

Figure 3.14: Widok grafu w interfejsie użytkownika z oznaczeniami: prostokąty - środki klastrów, kółka - węzły podrzędne, kółka z krawędziami będącymi pętlami - gatewaye. Kolory oznaczają przynależność do klastra.

Węzły kwadratowe reprezentują środkowe węzły poszczególnych klastrów. Węzły okrągłe stanowią węzły podrzędne klastra. Kolory odzwierciedlają podzialy na klastry. Zapętlone okrągłe węzły symbolizują Gatewaye.

3.6.4 Sposób implementacji

Implementacja modułu klasteryzacji znajduje się w pakiecie **pl.edu.agh.cs.kraksim.sna**. W głównym katalogu znajduje się interfejs **GraphVisualiser** udostępniający jedną metodę *refreshGraph()* oraz **SnaConfigurator** - klasa implementująca konfigurowanie parametrów metody.

Główna logika algorytmu zawarta jest w pakiecie **pl.edu.agh.cs.kraksim.sna.centrality**.

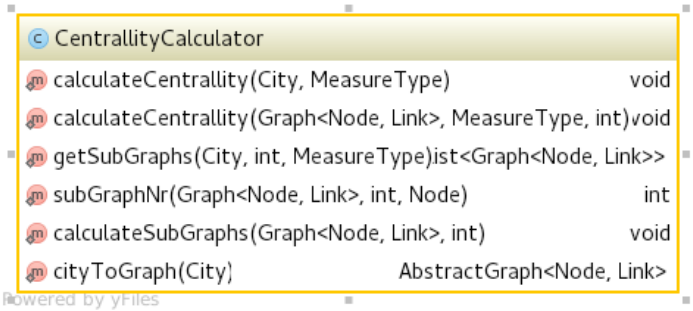

Figure 3.15: Klasa CentralityCalculator i jej publiczne metody

Pakiet ten składa się z sześciu klas:

CentralityCalculator Klasa pomocnicza obliczająca miary dla grafu. Jej najważniejszą metodą jest *calculateCentrality()*, która ma dwa warianty: z podaniem instancji **City** lub z bezpośrednim podaniem grafu. Jej drugim argumentem jest **MeasureType** - enum przyjmujący 3 wartości: *BetweenesCnetrality*, *PageRank* lub *HITS*. W zależności od podanej wartości metoda stosuje odpowiedni sposób liczenia wartości węzłów korzystając z biblioteki grafowej Jung.

Następnie metoda dokonuje podziału na podgrafy wywołując *calculateSubGraphs()* z argumentami typu **Graph** i liczbą podgrafów: *subGraphsNumber*. Potem następuje normalizacja miar w grafie i na koniec właściwa klasteryzacja przy użyciu metody *clusterGraph()* - statycznej metody z klasy **KmeansClustering**.

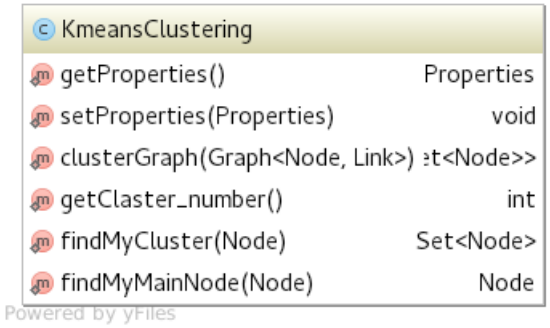

Figure 3.16: Diagram klasy KmeansClustering wraz z metodami publicznymi

KmeansClustering Klasa pomocnicza, która udostępnia jedną publiczną metodę statyczną służącą klastrowaniu wcześniej przygotowanego grafu metodą k-means.

Oprócz wymienionych klas w pakiecie znajdują się: **CentralityStatistics** - klasa pomocnicza do zapisywania wybranych statystyk, **OptimizationInfo** - klasa przechowująca informacje dotyczące preferencji zmian świateł dla skrzyżowania oraz pomocniczy enum **SNADistanceType**.

3.7 Moduł predykcji

Moduł sterowania sygnalizacją świetlną w systemie Kraksim podejmuje decyzję o wyborze odpowiedniej fazy sygnalizacji na podstawie informacji dostarczonych przez **moduł oceny**.

Ponieważ moduł oceny operuje jedynie na aktualnym stanie ruchu, logicznym jest stworzenie modułu, który mógłby informować o możliwości pojawienia się zagęszczeń zanim one rzeczywiście wystąpią.

W projekcie zaimplementowano więc moduł predykcji, który przewidując pojawienie się zatoru zmienia wartości ewaluacji dla każdego pasa o wartość związaną z tym przewidywaniem i odpowiednio dobraną wagą.

Informacje dostarczone przez moduł są wykorzystywane nie tylko przez sygnalizację, ale również przez kierowców. Ci korzystają co prawda z modułu przewidywania trasy, jednak predykcja taka informuje ich dużo wcześniej i może zapobiegać powstaniu korków zmieniając ich trasę.

3.7.1 Implementacja modułu predykcji

System predykcji wykorzystywany jest przez trzy elementy systemu:

- **sterownik symulacji** - wywołujący funkcję modułu w głównej pętli symulacji;
- **moduł routingu** - poprzez aktualizację czasu przejazdu przez daną ulicę;
- **moduł decyzyjny** - poprzez uwzględnianie predykcji przy zmianach fazy świateł

Sposób pozyskiwania danych

TODO

Główne elementy modułu

TODO

3.8 Generator ruchu

Generator ruchu nie jest w zasadzie oddzielnym modułem aplikacji w znaczeniu podanym wcześniej. Jednak stanowi na tyle ważny komponent, że zdecydowano go opisać oddzielnie.

3.8.1 Zasada działania

Aby wygenerowac ruch w symulacji definiuje się tzw. *schematy podróżowania*. Schemat dotyczy pewnej liczby samochodów i wyznacza kolejne węzły wlotowe, do których będą kierowały się samochody. Z każdym węzłem schematu związany jest czas wyruszenia z węzła, który zadany jest rozkładem losowym.

Działanie generatora sprowadza się do stworzenia samochodów dla wszystkich schematów podróżowania (w odpowiedniej krotności związanej ze schematem), a następnie umieszczania ich w modelu w turach wybranych na podstawie podanego przez użytkownika rozkładu.

3.8.2 Konfigurowalne parametry generatora

Mozliwe jest określenie zarodka używanego generatora liczb losowych *genSeed*.

Szczegóły dotyczące tworzenia schematów podróżowania opisane są w rozdziale 4. w sekcji Konfiguracja generatorów ruchu

3.8.3 Sposób implementacji

TODO

3.9 Moduł wyznaczania tras

Moduł wyznaczania tras przejazdu - nazywany czasami routerem, służy do wyznaczenia trasy przejazdu pomiędzy dwoma **Gatewayami**. Jak wspomniano w rozdziale opisującym interfejsy, implementacja modułu powinna definiować jedną metodę: *getRoute(Gateway, Gateway)*, która będzie zwracała obiekt typu **Route**.

3.9.1 Działanie routera

Dostarczona implementacja routera do wyznaczania tras przejazdu używa algorytmu Dijkstry. Samochód od jednego węzła (Gateway) do drugiego wybiera zawsze drogę po najkrótszej (względem długości krawędzi) trasie.

Generator ma również możliwość wyznaczania tras na podstawie tablicy średnich czasów przejazdu. Dane te dostarczane są w postaci kolekcji par (droga, czas). Tablica takich czasów budowana jest przez podsystem informacji o zatłoczeniach, na podstawie średnich danych z ustalonego okresu czasu. Dane te mogą powstawać w prosty statystyczny sposób: czasy przejazdu szacowane na podstawie średniej prędkości i długości drogi lub też mogą być aktualizowane przez moduł predykcji.

3.10 Moduł zbierania statystyk

3.10.1 Moduł Ministat

Moduł **Ministat** został stworzony przez mgr. inż. Batosza Rybackiego w ramach początkowych prac nad systemem Kraksim.

Jest dostępny w systemie do dziś w niemal niezmienionej formie.

Skojarzony jest on z trzema modułami:

- **CityMiniStatExt**

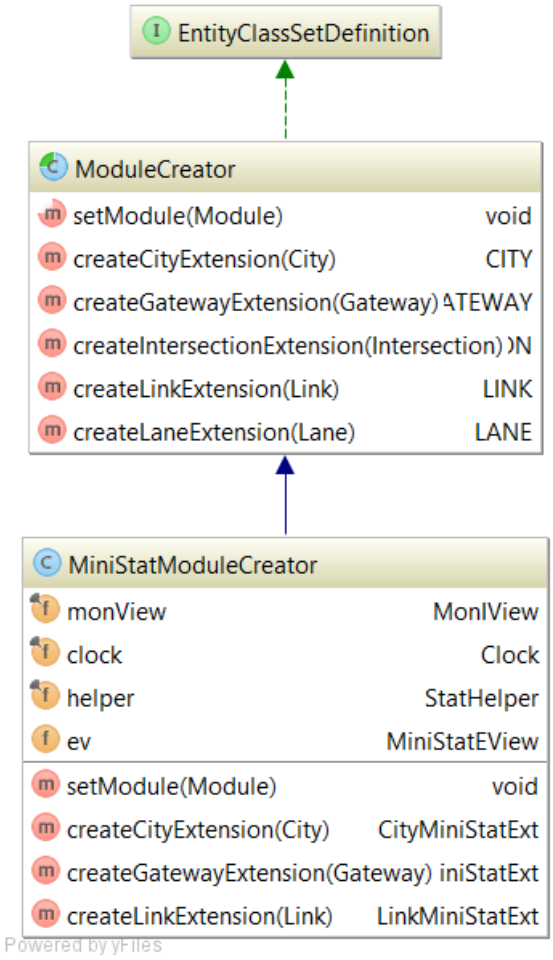

Figure 3.17: Diagram klasy **MinistatModuleCreator** wraz z przodkami

- **GatewayMiniStatExt**
- **LinkMiniStatExt**

Wymagany jest interfejs *Clock*, którego implementacją jest *Simulation*, służący do sprawdzania aktualnej tury.

Klasa StatHelper

Rozwnięcia Ministat korzystają z klasy **StatHelper** w celu zapamiętywania i przetwarzania danych statystycznych.

Udostępnia on następujące funkcjonalności:

- Zlicza ilość pojazdów
- Zlicza ilość przejazdów
- Zlicza długość przejazdów
- Zlicza czas przejazów
- Zlicza średnią prędkość pojazdów

Klasa CityMiniStatExt

Przetwarza dane globalne statystyczne, korzystając bezpośrednio z klasy **StatHelper**, oraz ma możliwość usunięcia wszystkich rozwinięć z instancji **Gateway** oraz **Link** w ramach miasta (**City**).

Klasa GatwayMiniStatExt

Przetwarza dane statystyczne podróży dla których punktami wejściowymi oraz wyjściowymi są instancje klasy **Gateway**. W tym celu, poprzez interfejs **GatewayMonIface** zakłada **CarEntranceHandler** na zdarzenia polegające na pojawianiu się nowych pojazdów na mapie. Następnie zbiera dane dotyczące przejazdów i zapamiętuje je w odpowiedniej strukturze danych.

Klasa LinkMiniStatExt

Ze wszystkich rozwinięć posiada najwięcej funkcjonalności. Przetwarza ona dane statystyczne jednego połączenia drogowego, w tym:

- Ilość samochodów wjeżdzających na odcinek
- Ilość samochodów opuszczających odcinek
- Średni czas przejazdów
- Średnią prędkość pojazdów

Dane te są pobierane i przetwarzane okresowo a następnie udostępniane przez interfejs.

Zastosowanie

Dane zbierane przez ten moduł są następnie wyświetlane za pomocą intancji klas *GUISimulationVisualizator* bądź *ConsoleSimulationVisualizator*.

Ponadto te same dane są pobierane przez metodę statyczną *dumpCarStats(...)* klasy *StatsUtil* i zapisywane za pomocą instancji klasy *PrintWriter* do pliku wyjściowego.

3.11 Dynamika systemu

W tej sekcji przedstawimy podstawowe etapy działania aplikacji tak by programista mógł umieścić działanie poszczególnych modułów w praktycznym kontekście.

Aby skupić uwagę czytelnika na aspekcie symulacji, nie uwzględniono w opisie działania modułu wizualizacyjnego i zbierania statystyk.

3.11.1 Inicjalizacja modułów

Poszczególne moduły inicjalizują się podczas powstawania autonomicznie. Wyjątkiem są moduły oceny i generatora ruchu, które muszą współdziałać z modułem fizycznym podczas rozpoczęcia działania.

Inicjalizacja modułu oceny

Moduł oceny rejestruje podczas inicjalizacji metody obsługi zdarzenia przejazdu samochodu w miejscu przez niego określonym. Do tego celu wykorzystuje metody znajdujące się w interfejsie **LaneMonIface**:

- *installInductionLoop()*
- *installInductionLoops()*

Inicjalizacja generatora ruchu

Wspomniano, że ruch samochodów w symulacji odbywa się na wyznaczonych trasach, które przebiegają przez kolejno zdefiniowane węzły (**Gateway**). W związku z tym generator ruchu musi reagować na zdarzenie dotarcia kierowcy do kolejnego węzła, ponieważ dopiero wtedy na bazie podanego rozkładu może wyznaczyć czas wyruszenia w kolejną (o ile dany węzeł nie jest już celem). Metody, które zapewniają tą funkcjonalność znajdują się w interfejsie **CitySimIface**.

- setCommonTravelHandler()

  ustawia wspólny dla wszystkich węzłów wyjazdowych obiekt obsługi zdarzenia końca podróży

- *setTravelEndHandler()*

  ustawia sposób obsługi zdarzenia zakończenia podróży do konkretnego węzła

3.11.2 Start symulacji

Podczas startu symulacji dokonuje się kilku czynności przygotowujących:

Ustawienie początkowego stanu modułu decyzyjnego

Sterownik symulacji nakazuje modułowi decyzyjnemu ustawienie początkowego stanu wywołując na nim metodę *initialize()* udostępnianego w ramach interfejsu **CityDecisionIface**. W trakcie ustawiania stanu początkowego moduł decyzyjny korzysta z metod z interfejsu blokowania pasów w celu ustalenia spójnego stanu pasów w całym mieście.

Generacja kierowców

Sterownik symulacji nakazuje modułowi generacji ruchu wygenerowanie kierowców zgodnie z implementowaną przez niego logiką.

3.11.3 Pętla symulacji

Tura symulacji

Pierwszym krokiem wykonywanym w ramach tury symulacji jest rozpoczęcie zaplanowanych na turę podróży.

Sterownik symulacji nakazuje generatorowi ruchu rozpoczęcie przeznaczonych na tę turę podróży. Generator ruchu przed umieszczeniem każdego pojazdu w mieście wyznacza dla niego trasę korzystając z metody *getRoute()*. Następnie wykorzystując metodę *insertTravel()* z interfejsu **CitySimIface**, samochód umieszczany jest w symulacji.

Następnym krokiem jest symulacja jazdy samochodów w ramach tury. Na tym etapie pojazdy przemieszczają się w modelu miasta. Etap ten odbywa się w module fizycznym poprzez wywołanie metody *simulateTurn()* interfejsu **CitySimIface**.

3.11.4 Aktualizacja oceny dla pasów

Po zakończeniu tury symulacji, sterownik symulacji wywołuje na module oceny metodę *turnEnded()* (interfejs **CityEvalIface**). W ramach tej metody moduł oceny ma za zadanie uaktualnić oceny dla wszystkich pasów.

Następnie ostatnim etapem jest poinformowanie modułu decyzyjnego o końcu tury. Służy do tego metoda *turnEnded()* z interfejsu **DecisionEvalIface**. Moduł decyzyjny pobiera informację oceny pasów korzystając z metody: *getEvaluation()* oraz *getMinGreenDuration()* z interfejsu **LaneEvalIface**.

Następnei dokonuje oceny zgodnie ze swoją implementacją i wywołuje odpowiednie metody z interfejsu blokowania pasów.

TODO: dodać predykcję do dynamiki

3.11.5 Rozruch aplikacji

KraksimRunner

Ta klasa posiada metodę *main(String[])*.Oczekiwany jest tylko jeden argument - nazwa pliku konfiguracyjnego.

Wykonywane są następujące kroki:

1. Nazwa pliku konfiguracyjnego jest następnie wykorzystywana do pozyskiwania konfiguracji za pomocą klasy *KraksimConfigurator*. Jeżeli nie podano ścieżki do pliku konfiguracyjnego, używana jest domyślna ścieżka.
2. Z konfiguracji pobierana jest zmienna konfiguracyjna decydująca o obecności vizualizacji.
3. Z konfiguracji pobierana jest zmienna konfiguracyjna decydująca o obecności predykcji w symulacji.
4. Jeżeli aplikacja skonfigurowana jest do wizualizacji, inicjowane jest środowisko graficzne. Wygląd konfigurowany jest z pomocą klasy Swing'owej *UIManager*, która za argument przyjmuje ustawienia w postaci łancuch znaków (*String*).
5. Uruchomiony zostaje wątek symulacji. Konfiguracja dla niego jest budowana za pomocą klasy *KraksimConfigurator* przy użyciu metody *prepareInputParametersForSimulation()*.

KraksimConfigurator

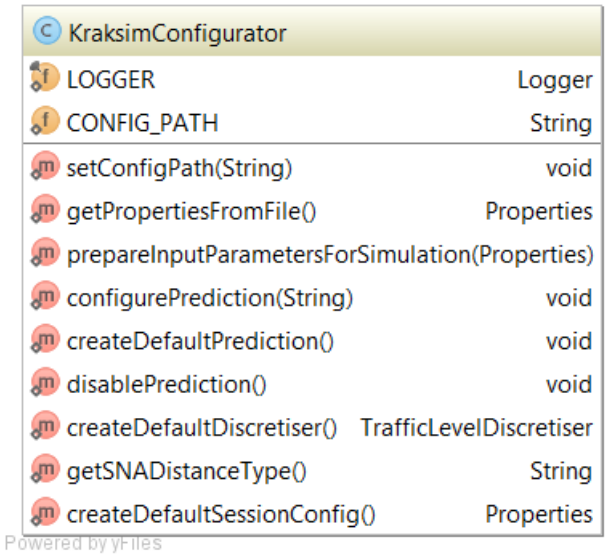

Figure 3.18: Diagram klasy KraksimConfigurator

Jej zadaniem jest pobieranie konfiguracji z pliku konfiguracyjnego i udostępnianie jej za pomocą wygodnego interfejsu.

Konfigutator wymaga ścieżki do pliku konfiguracyjnego, który musi być ustawiony przez metodę *setConfigPath(String confingPath)*. Jeżeli tak się nie stanie to domyślną ścieżką do pliku konfiguracyjnego jest: *"configuration/kraksim.properties"*.

Wynikiem działania konfiguratora jest lista opcji o nazwie *paramsList*, w postaci *List<String>*, który następnie jest używany jako parametr wywołania symulacji. Format jest listy został opisany w rodziale zawierającym postać plików wejściowych i wyjściowych, jako że ze względu na jej prostotę i format tekstowy a także aspekty historyczne - kiedyś lista ta była listą argumentów przy wywołaniu systemu Kraksim z linii poleceń i ze względu na kompatybilność pozostała w tej formie. Warto rozważyć modyfikację tego rozwiązania.

3.12 Klasa Simulation

Klasę simulation można uznać za najważniejszą w całej aplikacji. Jej role:

1.

3.13 Architektura interfejsu graficznego

Do budowy graficznego interfejsu użytkownika w projekcie Kraksim wykorzystano bibliotekę Swing.

3.13.1 Swing

Swing jest narzędziem dla języka Java, powstałym na bazie AWT, upublicznionym w 1997 roku. Była ona powszechnie stosowanym narzędziem do tworzenia przez programistów graficznych interfejsóœ użytkownika. Jej zaletami jest sprawdzone działanie i zapewnienie, że aplikacja działająca na różnych środowiskach będzie dostosowała się wyglądam do platformy, na której się znajduje. Obecnie rzadko można spotkać pisanie nowych interfejsów użytkownika przy pomocy tej biblioteki. Do jej wad można zaliczyć utrudnionione dostosywywanie wyglądu i struktury GUI jak również niespójne API.

3.13.2 Klasy implementujące interfejs użytkownika

Klasy implementujące interfejs to: **MainVisualizationPanel.java** oraz **SetUpPanel.java**. Są one inicjalizowane w momencie uruchomienia aplikacji.

MainVisualisationPanel

Definiuje 4 metody publiczne:

- **MainVisualizationPanel(Properties)** - konstruktor - inicjalizuje parametry symulacji, layout okien i klika przycisk load. Najważniejsza metoda prywatna jaką wywołuje to **initLayout()** - jest ona odpowiedzialna za stworzenie całego layoutu interfejsu graficznego oraz instancjonowaniu panelu konfiguracyjnego.
- **initializeSimulation(Properties):void** - metoda ustawiająca parametry symulacji, tworząca obiekt symulacji Simulation oraz obiekt wizualizacji symulacji SimulationVisualizator. Na końcu uruchamia symulację w osobnym wątku.

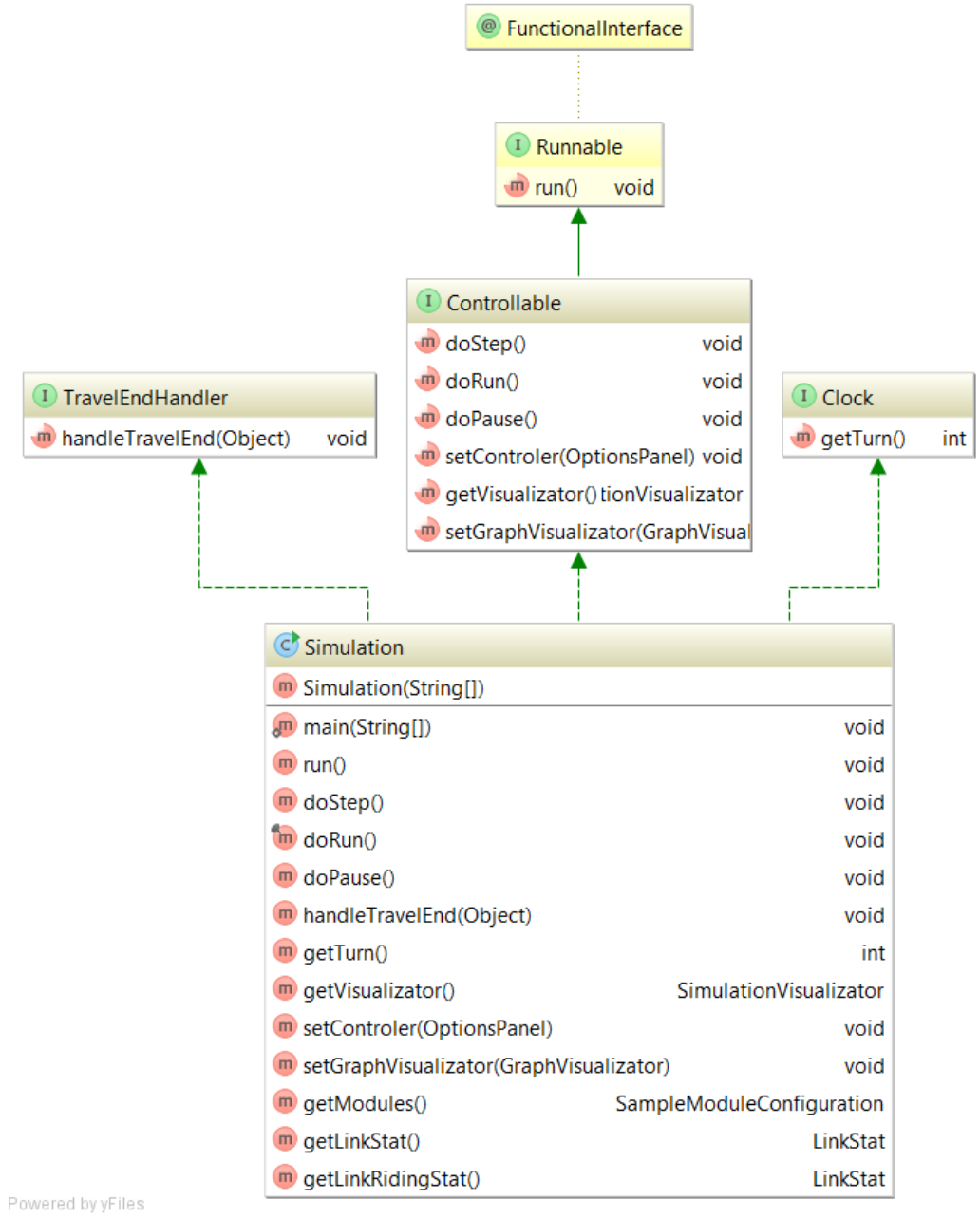

Figure 3.19: Diagram klasy Simulation z uwzględnieniem tylko metod

- **getSimulation():Simulation** - getter zwracający obiekt typu Simulation
- **refreshGraph():void** - metoda odświeżająca stan grafu. Wylicza centrality, odświeża odległości pomiędzy węzłami w ramach metryki, dostowuje kolory w grafie.

SetUpPanel

Definiuje 4 metody publiczne:

- **SetUpPanel(MainVisualizationPanel, Properties)** - podstawowy konstruktor klasy. Inicjalizuje layout okna konfiguracyjnego ze wszystkimi jego funkcjonalnościami
- **SetUpPanel(MainVisualizationPanel)** - drugi konstruktor klasy. Obecnie nie używany w aplikacji. Wykonuje tą samą pracę co pierwszy z tą różnicą, że nie ustawia podanych Properties, a inicjalizuje nowy, pusty obiekt Properties.
- **initLayout():void** - metoda wywoływana z konstruktora SetUpPanel. Tworzy okno konfiguracji
- **end():void** - kończy inicjalizację layoutu i klika przycisk "Load" odpowiedzialny za załadowanie parametrów z okna kofiguracyjnego do symulacji.

Rozdział 4

Formaty plików wejściowych i wyjściowych

4.1 Format listy konfiguracyjnej symulacji

Składania pliku konfiguracyjnego bazuje na oryginalnej składnie wywołania aplikacji z (Rybacki2008):

```
kraksim [options] algConf modelFile trafficFile
algConf selects and configures traffic light system driver.
syntax: algConf = algName[:param1=val,param2=var,...]
options:
-v: turns on visualization (default: on)
-g: turns on visualization (default: off)
-z: turns on zone awareness (default: off)
-X model: sets the car move model (default: nagle)
-Q module: sets the basic implementation of move module
-Z switch_time:
-e: enables prediction (default: false)
-a prediction_module: sets the prediction model (dault: false)
-Y minimal_safe_distance: sets minimal safe distance
-t transitionDuration : sets the duration of traffic lights transitional state (de
-s modelSeed : sets the seed of the traffic simulator RNG (default: based on the sy
-S genSeed: sets the seed of the traffic generator RNG (default: based on system cl
-o statFile: statistics file name (default: no statistics are generated)
-r: enables dynamic routing (driver canchange route)
-k isRoutingTh: sets the percentage level of drivers that are willing to change rou
-u interval: sets the interval for time table update process
-m speed: minimal speed when using prediction
```

4.2 Format pliku wejściowego

Pliki wejściowe dla systemu symulacji ruchu drogowego mają postać strukturalnych dokumentów xml

4.2.1 Sieć drogowa

Aby opisać konfigurację sieci drogowej należy utworzyć dokument xml z następują- cym nagłówkiem (standardowy nagłówek określający xml, format 1.0). Ten dokument zaczyna się od tagu *RoadNet*.

```
<?xml version="1.0"?>
<RoadNet>
```

Plik konfiguracji schematu sieci drogowej powinien zawierać informacje o rozmieszcze- niu węzłów sieci, informacje opisujące parametry dróg (długość, pasy zjazdowe) oraz informacje opisujące skrzyżowania. Proces budowy modelu sieci drogowej wymusza określoną kolejność tych trzech bloków opisujących daną sieć drogową. Najpierw należy opisać węzły i ich położenie. Jest dwa rodzaje węzłów gateway - węzeł wejścia/wyjścia oraz intersection - skrzyżowanie.

```
<nodes>
<gateway id="N" x="500" y="10"/>
<gateway id="E" x="990" y="500"/>
<gateway id="S" x="500" y="990"/>
<gateway id="W" x="10"
y="500"/>
<intersection id="intersect" x="500" y="500"/>
</nodes>
```

Drugi blok opisuje ulice. Każda ulica (a właściwie każdy fragment łączący dwa węzły) musi mieć unikalny identyfikator, oraz pola from i to, które określają kierunek ulicy oraz węzły(w tym skrzyżowania) będące dwoma końcami ulicy. Poniższy przykład zawiera opis drogi, która bieg- nie od węzła N do węzła intersect (kierunek jest ważny dla ustalenia który pas jest lewy, a który prawy). Pas (połączenie), którym samochody poruszają się w kierunku z N do intersect to uplink, i ma długość 1000. Natomiast downlink to połączenie, którym pojazdy poruszają się w kierunku przeciwnym czyli z intersect do N. Identyfikatorem tego fragmentu drogi jest *Nroad*, a jej nazwą (oraz nazwę całej drogi do której należy ten fragment, nazwa drogi w przeciwieństwie do identy- fikatora nie musi być unikalna i nie jest obowiązkowa) jest ulica Pionowa.

```
<roads>
<road id="Nroad" street="ulica Pionowa" from="N" to="intersect" >
<uplink>
<main length="100"/>
</uplink>
<downlink>
```

```
<main length="100"/>
130
A.3. Program
</downlink>
</road>
```

Definiując drogę należy określić, czy ma ona dodatkowe pasy do skrętu w lewo i/lub w prawo. Następny przykład zawiera opis drogi prawie identyczny jak ten wyżej. W tym przypadku połączenie biegnące z N do intersect posiada dodatkowo pas do skrętu w lewo o długości 5 jednostek (w naszej symulacji jednostka to 7,5 metra), a połączenie w przeciwnym kierunku posiada dodatkowo pasy do skrętu w lewo jak i w prawo, obydwa o długości 5 jednostek.

```
<road id="Nroad" street="ulica Pionowa" from="N" to="intersect" >
<uplink>
<main length="100"/>
<left length="5"/>
</uplink>
<downlink>
<right length="5"/>
<main length="100"/>
<left length="5"/>
</downlink>
</road>
</roads>
```

Ostatni - trzeci blok opisowy to schemat budowy i działania skrzyżowania.

```
<intersectionDescriptions>
```

Zawiera on identyfikator skrzyżowania, oraz listę akcji dla każdego ramienia.

```
<intersection id="intersect">
```

Każda akcja składa się z reguł nadrzędności.

```
<armActions arm="Nroad">
```

Dla jednego połączenia wychodzącego ze skrzyżowania (dla danego ramienia) definiowana jest jedna akcja. Jedna akcja może zawierać od 0 do n reguł nadrzędności (zawierają one nazwy pasów nadrzędnych dla bieżącego), gdzie n jest mniejsze od liczby pasów wchodzących do skrzyżowania.

```
<action lane="0" exit="Eroad">
<rule entrance="Sroad" lane="0"/>
</action>
<action lane="0" exit="Sroad"/>
<action lane="0" exit="Wroad"/>
```

```
</armActions>
<armActions arm="Eroad">
<action lane="-1" exit="Sroad">
<rule entrance="Nroad" lane="0"/>
<rule entrance="Sroad" lane="0"/>
<rule entrance="Wroad" lane="0"/>
</action>
<action lane="0" exit="Wroad">
<rule entrance="Nroad" lane="0"/>
<rule entrance="Sroad" lane="0"/>
</action>
<action lane="0" exit="Nroad">
<rule entrance="Sroad" lane="0"/>
</action>
</armActions>
<armActions arm="Sroad">
[...]
</armActions>
```

Akcja dotyczy jednego pasa - *lane*(może to być np. pas do skrętu w lewo). Akcja ma podany cel (tutaj nazwany *exit*). Taka akcja określa z którego pasa ruchu i w którym kierunku możliwy jest ruch pojazdu. Każda reguła zawiera numer dokładnie jednego pasa jednej drogi(ulicy), który ma wyższy priorytet niż pas którego dotyczy dana akcja. Opis skrzyżowania powinien zawierać listę faz sygnalizacji świetlnej.

```
<phase num="1" duration="20" name="NS">
<inlane arm="Nroad" lane="0" state="green" />
<inlane arm="Nroad" lane="-1" state="red" />
132
A.3. Program
<inlane arm="Sroad" lane="0" state="green" />
<inlane arm="Sroad" lane="-1" state="red" />
<inlane arm="Wroad" lane="0" state="red" />
<inlane arm="Wroad" lane="-1" state="red" />
<inlane arm="Eroad" lane="0" state="red" />
<inlane arm="Eroad" lane="-1" state="red" />
</phase>
<phase ...>
[...]
</phase>
```

Opis jednej fazy sygnalizacji składa się z listy wszystkich pasów wchodzących do skrzyżowania wraz z kolorami świateł dla tych pasów. Każda faza ma swój numer.

```
<plan name="WE" >
<phase num="1" duration="20" name="NS" />
<phase num="2" duration="10" name="lNS" />
<phase num="3" duration="68" name="WE" />
<phase num="4" duration="10" name="lWE" />
</plan>
<plan name="NS" >
<phase num="1" duration="68" name="NS" />
<phase num="2" duration="10" name="lNS" />
<phase num="3" duration="20" name="WE" />
<phase num="4" duration="10" name="lWE" />
</plan>
</intersection>
</intersectionDescriptions>
```

Na zakończenie pliku wpisujemy tag:

```
</RoadNet>
```

Pełne przykłady plików dostarczone są wraz z projektem w zestawach testowych. Do zestawów, dołączone są krótkie opisy, rysunki i wykresy. Jeśli ten opis czegoś nie wyjaśnia to na pewno można to zrozumieć z zestawów testowych.

4.2.2 Konfiguracja generatorów ruchu

Konfiguracja generatorów ruchu, czyli opis wzorów generowania pojazdów dla potrzeb symulatora jest prostszy niż budowa sieci dróg. Plik zaczyna się podobnie, na początku nagłówek xml, wersja 1.0 oraz tag rozpoczynający dokument, w tym przypadku <*traffic*>.

```
<?xml version="1.0"?>
<traffic>
```

Główną część tego pliku stanowią schematy ruchu. Pojedynczy schemat posiada atry- but określający ilość pojazdów – count. Schemat to po prostu lista węzłów (typu gateway) wejściowych, z wyszczególnieniem pory pojawienia się pojazdów.

```
<scheme count="250">
<gateway id="N">
```

Czas pojawienia się pojazdów określa się podając jeden z trzech możliwych rozkładów, oraz jego parametry. Dostępne rozkłady to rozkład normalny, rozkład jednorodny oraz dokładny punkt czasu, w którym grupa pojazdów ma się pojawić.

```
<uniform a="0" b="2700"/>
</gateway>
```

Aby zdefiniować rozkład punktowy wystarczy podać tylko punkt w jednowymiarowej przestrzeni czasu y (jednostką jest sekunda, początek jest o północy - początek doby).

```
<point y="1200"/>
```

Rozkład normalny określa się podając czas *y* , oraz wartość odchylenia standardowego *dev*:

```
<normal y="1200" dev="30"/>
```

Rozkład jednostajny określa się podając punkt startu i końca *a* i *b*.

```
<uniorm a="1000" b="1300"/>
```

Należy pamiętać że ostatni węzeł w każdym schemacie nie posiada informacji o czasie, gdyż jest to węzeł docelowy – wyjściowy i w nim nie generuje się pojazdów do symulacji. W ostatnim węźle pojazdy są „niszczone”.

```
<gateway id="E">
</gateway>
134
A.3. Program
</scheme>
<scheme count="250">
<gateway id="N">
<point y="200"/>
</gateway>
<gateway id="S">
</gateway>
</scheme>
<scheme count="1000">
<gateway id="N">
<normal y="2700" dev="20"/>
</gateway>
<gateway id="W">
</gateway>
</scheme>
</traffic>
```

4.3 Format pliku wyjściowego

4.3.1 Pliki wygnerowane przez moduł ministat

W ramach początkowego rozwoju aplikacji, dla potrzeb pracy magisterskiej mgr. inż. Bartosza Rybcakiego zaimplementowano moduł statystyk *MiniStat*. W trakcie symulacji generowane są statystyki

szczegółowe i za pomocą klasy pomocniczej *SatsUtil* są one zapisywane do pliku tekstowego z rozszerzeniem *txt*. Po zakończeniu symulacji obliczane są statystyki sumaryczne i zapisywane są do drugiego pliku statystyk, tym razem z rozszerzeniem *txt.sum*. Pliki statystyk zawierają wartości odzielone znakami odstępu (*csv*).

Format pliku statystyk szczegółowych

Plik statystyk szczegółowych zaczyna się nagłówkiem:

```
<LINK_1> <LINK_2> ... <LINK_N> #of_travels #of_cars avg_velocity
```

Kolejne pola <*LINK_1*> w nagłówku to nazwy skierowanych połączeń, wartości w kolejnych wierszach odpowiadające tym nagłówkom pokazują liczbę pojazdów, które przejechały przez dane połączenie. *#of_travels* oznacza ilość zakończonych podróży. *#of_cars* to ilość pojazdów w systemie symulacji, czyli ilość pojazdów, które aktualnie przemierzają drogi modelu zsumowana z ilością pojazdów, które już wystartowały, ale jeszcze nie są obecne w modelu, ponieważ czekają w kolejce za punktem wjazdu (ta wartość pomaga porównać różne przebiegi symulacji). *avg_velocity* oznacza średnią prędkość w systemie.

Table 4.1: My caption

...	I85I81	I5I39	I85I83	I48A14	#of_travels	#of_cars	avg_velocity
...	0	0	0	0	0	103	0
...	0	0	0	0	3	188	1.5816326
...	0	7	0	0	15	274	1.579235
...	0	8	2	0	34	355	1.5075688
...	0	11	4	1	71	401	1.4653335
...	0	13	5	4	117	440	1.431609
...	0	17	6	5	173	481	1.3891101

Przykład:

Format pliku statystyk sumarycznych

Ten plik podzielony jest na 3 części. Pierwsza części zawiera czas trwania symulacji w turach oraz średnią prędkość.

```
CITY STATS
==========
sim. duration avg. velocity
43201 1,23
```

Druga część zawiera statystyki tras (gdzie trasa rozumiana jest jako podróż z miejsca początkowego do celu). W każdej linijce pojawiają się następujące pola:

- identyfikator węzła źródłowego
- identyfikator węzła docelowego
- liczba podróży
- średni czas trwania podróży w turach
- odchylenie standardowe
- średnia prędkość dla tej trasy wyrażona w komórkach na turę oraz w kilometrach na godzinę

4.3.2 Pliki statystyczne dodane przy pracy Dygon i Kawula

```
ROUTE STATS
==========
from
to count avg. duration <-std dev. avg. velocity
<-[kph]
G4 G3 616 561,0 650,2 0,71 19,3
G4 G6 609 954,5 638,6 1,10 29,7
```

Trzecia część zawiera statystyki dla połączeń. Znaczenie kolejnych pól jest dokładnie takie samo jak, dla statystyk tras. Oczywiście w tym przypadku tabela zawiera statystyki dla wszystkich połączeń modelu, a nie dla tras.

```
LINK STATS
==========
from
to count avg. duration <-std dev. avg. velocity
<-[kph]
G6 R6 5842 56,5 1,7 1,77 47,8
X12 X9 3854 341,4 26,7 1,61 43,5
```

- wypisywanie statystyk
  - vote_stat/*ŧxt - przechowuje miary wyznaczone przez każde ze skrzyżowań wobec wszystkich pozostałych skrzyżowań wykorzystywane do głosowania.
  - centrallity_stat/clusteringData(number)ŧxt przechowuje skrzyżowania główne wraz z ilością głosów jakie otrzymały w turze number.
  - centrallity_stat/travelTimes(data)ŧxt - przechowują informację: tura + czas podróży + średnią prędkość + średnią szybkość + średnie obciążenie + łączny czas podróży do celu + ilość samochodów.
- wypisywanie logow
  - speed.log - przechowuje turę + średnią prędkość dojazdu samochodu do celu
  - carsOnRed.log - przechowuje ilość samochodów stojących na czerwonym świetle
  - clustering.log - przechowuje informacje, które skrzyżowania zostały wybrane na skrzyżowania główne i kto na nie głosował
  - calculator.log - przechowuje wagi krawędzi grafu (połączenia między skrzyżowaniami) wyliczonymi przy pomocy page rank lub betweenness centrality
  - fullLog.log - przechowuje wszystkie logi

    Wszystkie dane wyjściowe zapisywane są w folderze “output”. Logi skatalogowane są w podfolderze “logs”.

Rozdział 5

Podsumowanie

Uważamy, że ta dokumentacja pomogła państwu w rozwijaniu projektu Kraksim.

5.1 Dodatkowe informacje

Jeżeli chcą państwo poznać informację odnośnie użykowania aplikacji:

1. Dokumentacją użytkownika

Dodatek A

Historia programu Kraksim

A.1 Funkcjonalności dodane w poszczególnych wersjach produktu

1. Matyjewicz i Rybacki (2008)
   - opracowanie prototypu
   - budowa symulatora/optymalizatora ruchu miejskiego
   - implementacja modułów: sieci drogowej, ruchu, sterownika sygnalizacji, oceny sytuacji i decyzyjnego
2. Adamski i Pierzchała
   - generowanie ruchu na podstawie rzeczywistych danych
3. Dziewoński i Zalewski (2008)
   - Dodanie obslugi wielopasowości
   - Dodanie funkcjonalności predykcji korków
4. Kurek i Rajczyk
   - Optymalizacja ruchu drogowego poprzez klasteryzacje opartą na miarach SNA
5. Kot i Skałka (2009)
   - Dodanie kreatora mapy w Swingu
   - Przeprowadzone testy
   - Dodanie statystyk średnich prędkości na drogach
   - Dodanie w konfiguracji atrybutu minimal speed
   - Drobne zmiany: kolorowanie pojazdów, zapisywanie ustawień

6. Doliński (2012)
    - Dodanie modułu wykrywania i zapobiegania zagęszczeniom ruchu drogowego w systemie Kraksim
7. Legień i Bibro - Systemy Agentowe (2013)
    - Dodanie algorytmu głosowania na wielu kandydatów (+ dodatkowy panel w GUI)
    - Wykonanie analiz efektywności algorytmów w zależności od parametrów
8. Małgorzata Majcherek, Aleksander Lech (2012/2013)
    - Lepszy opis korków
    - Randomizacja zależna od prędkości (VDR)
    - Model świateł stopu (BL)
    - Próba implementacji wielopasmowego modelu Nagla-Schreckenberga
9. Bober i Liput (2014)
    - Połączenie powyższych wersji
    - Rozbudowa opisu dróg
    - Zmiana alogrytmu wyznaczania tras
    - Nowy generator pojazdów
    - Refaktoring kodu

    Propozycje zespołu

    - Poprawienie mapy (ulice jednokierunkowe)
    - Dalszy refaktoring
    - Dodanie obsługi błędów na poziomie UI
10. Haręza i Ostaszewski (2014)
    - moduł wizualizacyjny
11. Skurnóg i Wawryka (2015)
12. Dygoń i Kawula (2015)
    - Integracja systemu symulacji ruchu z modułami wizualizacyjnymi
    - Przywrócenie działania modułu wyboru głównych skrzyżowań i klasteryzacji
    - Dodanie nowego sposobu liczenia odległości między skrzyżowaniami (nieeuklidesowej)
    - Wypisywanie statystyk i logów

- Poprawa błędów

13. Dygoń i Kawula (2015)
    - Integracja systemu symulacji ruchu z modułami wizualizacyjnymi
    - Przywrócenie działania modułu wyboru głównych skrzyżowań i klasteryzacji
    - Dodanie nowego sposobu liczenia odległości między skrzyżowaniami (nieeuklidesowej)
    - Wypisywanie statystyk i logów
    - Poprawa błędów
    - Testowanie algorytmów

    Propozycje zespołu

    - Głosowanie z sumieniem
    - Głosowanie z punktacją
    - Głosowanie z dzieleniem głosów
    - Testy metod głosowań

Dodatek B

Bibliografia

1. "Modelowanie i optymalizacja ruchu miejskiego przy użyciu wybranych technik" - Bartosz Rybacki
2. "Kraksim - edytor planu miasta, nowa mapa, dostosowywanie ruchu do rzeczywistości, wymuszanie minimalnych prędkości - Dokumentacja do projektu" - Krzysztof Kot, Sławomir Skałka
3. "Kraksim – wielopasowość i predykcja ruchu - Dokumentacja do projektu" - Maciej Zalewski, Łukasz Dziewoński
4. "Rozbudowa i integracja różnych modeli ruchu w środowisku symulacji ruchu drogowego (Kraksim)" - Bober Piotr, Liput Jakub
5. "Systemy agentowe Generowanie ruchu w systemie Kraksim na podstawie danych z ZDKiA" - Tomasz Adamski,Paweł Pierzchała
6. "Kraksim - algorytm Reinforcement Learning with Context Detection i algorytm zapewniania minimalnych prędkości na drogach" - Piotr Doliński, Michał Skowron
7. "Implementacja algorytmu opartego na miarach SNA do optymalizacji ruchu drogowego Modyfikacje systemu Kraksim" - Maciej Kurek, Grzegorz Rajczyk
8. "Elastyczne środowisko do modelowania i optymalizacji ruchu drogowego" - Piotr Doliński
9. "Wybór głównych skrzyżowań w oparciu o głosowanie" - Grzegorz Legień, Jakub Bibro
10. "Integracja istniejącego systemu do symulacji ruchu drogowego w mieście z modułami wizualizacyjnymi - DOKUMENTACJA PROJEKTU" - Marcin Haręza, Michał Ostaszewski
11. "Dokumetnacja do projektu" - Jonasz Dygoń Andrzej Kawula
12. "Rozbudowa środowiska do modelowania i optymalizacji ruchu drogowego o różne algorytmy modelowania ruchu." - Małgorzata Majcherek, Aleksander Lech