Dokumentacja Agenty 007: DoCelu

Piotr Pakulski, Maciej Bekas, Artur Grudkowski, Patrycja Karbownik, Wojciech Wolny

Grudzień 2021

Spis treści

1 Etap A

1.1 Problem

Małe miejscowości mają mało pojazdów transportu publicznego i stałą dużą pulę przystanków autobusowych. Autobusy jeżdżą bardzo rzadko i przez wiele przystanków są puste, co nie opłaca się zarządcy autobusów, czego wynikiem jest usuwanie coraz większej liczby linii autobusowych w tychże miejscowościach. To powoduje wykluczenie komunikacyjne dla wielu mieszkańców. [1], [2], [3]

Innym, równie istotnym problemem jest brak nocnych połączeń.

1.2 Interesariusze

Interesariuszy naszego problemu podzieliliśmy na dwie kategorie. Pierwszą są interesariusze pierwszoplanowi, którzy bezpośrednio odczują skutki naszego potencjalnego rozwiązania problemu. Druga kategorią są interesariusze drugoplanowi, na których w pośredni sposób może wpłynąć nasze rozwiązanie. Poniżej przedstawiamy wybrane przez nas podmioty:

1. Pierwszoplanowi:
   (a) Mieszkańcy miejscowości - wpływ pozytywny
   (b) Zarządca autobusów - wpływ pozytywny
   (c) Kierowcy - wpływ pozytywny
   (d) Osoby starsze (wykluczone technologicznie) - wpływ negatywny
2. Drugoplanowi:
   (a) Inni lokalni przewoźnicy - wpływ negatywny
   (b) Dostawcy paliwa - wpływ negatywny
   (c) Mechanicy - wpływ neutralny

Głównymi beneficjentami naszego rozwiązania będą przede wszystkim mieszkańcy wsi ze względu na szybszą i łatwiejszą komunikacje. Dodatkowo do podmiotów, które skorzystają na naszym rozwiązaniu, należy zaliczyć zarządców pojazdów (busiki/autobusy), ze względu na oszczędności wynikające z optymalnego zarządzania pojazdami, oraz kierowców, którzy będą mieli częstsze postoje i większe bezpieczeństwo ze względu na kontrolę osób korzystających z pojazdów. Osoby starsze i wykluczone technologicznie mogą natomiast najbardziej stracić na tym rozwiązaniu. Planujemy jednak zaproponować dodatkowe rozwiązanie, które będzie miało na celu pomoc takim osobom w korzystaniu z transportu.

Do kategorii interesariuszy drugoplanowych sklasyfikowaliśmy innych lokalnych przewoźników, którzy przed wprowadzeniem naszego rozwiązania mogą zarabiać dzięki niedostosowanemu grafikowi głównego zarządcy autobusów. Dodaliśmy do tej kategorii również dostawców paliw ze względu na potencjalne ograniczenie zużycia paliwa w miasteczku. Ostatnim podmiotem zostali mechanicy z założeniem, że mogą mieć więcej lub mniej pracy zależnie od podejścia zarządcy autobusów do zaoszczędzonych pieniędzy. Z jednej strony można zainwestować w częstsze i bardziej kompleksowe naprawy pojazdów, z drugiej zarządca może zainwestować oszczędzone pieniądze w inny sposób.

1.3 Potencjalne rozwiązania

- Autobusy na życzenie (aplikacja) - osoba wyznacza przystanek początkowy i docelowy, a system tak dobiera trasę przejazdu autobusu, aby była ona jak najwydajniejsza zarówno dla autobusu jak i pasażerów, przy minimalizacji czasu oczekiwania na autobus i maksymalizacji zajętości miejsc w autobusie.
- Autonomiczne auta (Uber) - po danym obszarze krąży lub jest rozlokowana w strategicznych punktach flota autonomicznych aut, które są gotowe do transportu pasażerów po okolicy.
- Cyfryzacja wszystkiego - zapewnienie wszystkich potrzeb bez wychodzenia z domu.
- Lokalny Car-sharing (a'la BlaBlaCar) - osoba oznacza gdzie jedzie i kiedy wraca, informacja jest udostępniona notyfikacją push użytkownikom, którzy mieszkają przy trasie w określonej odległości od niej. Użytkownik sam wybiera osoby, którym udostępni trasę (wszystkim/przyjaciołom). Do tego aplikacja proponuje dodanie do znajomych osób, które mieszkają na trasie.

1.4 Nasze rozwiązanie

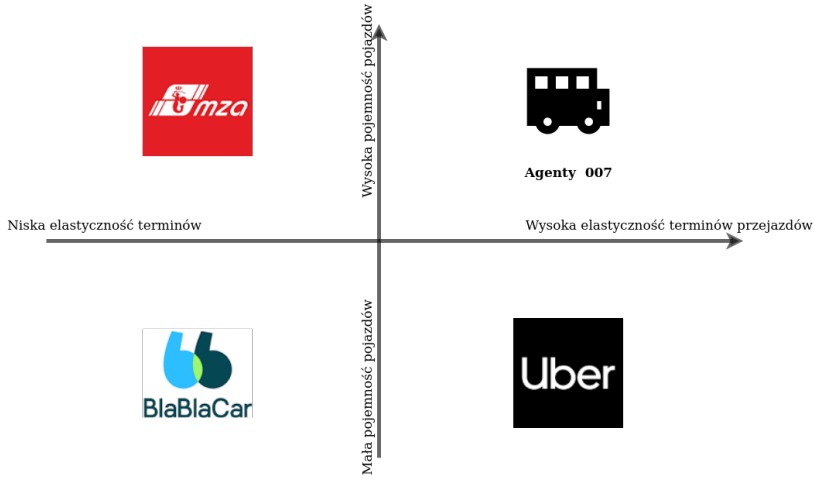
Aplikacja do zamawiania pojazdów różnej wielkości na godzinę/teraz do przemieszczania się możliwie blisko określonego celu. Dodatkowo osoby prywatne mogą rejestrować swoje auta jako środek transportu na określonej przez siebie trasie. Autobusy nieużywane zostają równomiernie rozmieszczone na terenie miejscowości w celu optymalizacji czasu oczekiwania na autobus. Jest to swoiste połączenie potencjalnych rozwiązań mające na celu maksymalizację wykorzystania istniejącej już infrastruktury dzięki czemu koszty jak i czas wdrożenia będą jak najniższe.

1.5 Analiza źródeł

Istnieje wiele rozwiązań, które w mniejszym lub większym stopniu rozwiązują postawiony problem. Po przeprowadzonej analizie źródeł najbliższym do naszego rozwiązania wydaje się platforma Zeelo, zapewniająca elastyczne programy transportowe. Jednak jej ograniczeniem jest wykorzystanie jedynie autobusów jako środka transportu co sprawia, że nie zawsze jest to optymalne rozwiązanie w przypadku mniejszych miast. Dodatkowo domyślnie w platformach tego typu autobusy czy też inne pojazdy dostarcza jedna firma co jest też swego rodzaju ograniczeniem i pozwala na pewne nadużycia np. dyktowanie cen. Nasze rozwiązanie rozwiązuje oba problemy udostępniając możliwości świadczenia usług dla każdej osoby/firmy posiadającej uprawnienia i umożliwiając użycie wielu środków transportu.

Lista podobnych rozwiązań:

1. Zeelo - `https://www.uktech.news/news/uk-uber-for-buses-zeelo-funding-20210811` - inteligentna platforma autobusowa dla organizacji zapewniająca elastyczne programy transportowe.
2. BlaBlaCar - `https://www.blablacar.pl/` - Serwis do wspólnych przejazdów szczególnie dla osób podróżujących na długich dystansach.
3. Uber - `https://www.uber.com/` - Aplikacja do zamawiania transportu samochodowego poprzez łączenie pasażerów bezpośrednio z kierowcami korzystającymi z aplikacji.
4. Lyft - `https://www.lyft.com/` - Podobnie jak Uber, aplikacja oferująca usługi transportowe poprzez łączenie pasażerów bezpośrednio z kierowcami, a także oferująca wynajem różnego rodzaju pojazdów.
5. Panek - `https://panekcs.pl/` - Aplikacja oferująca wynajem samochodów na minuty.
6. Operatti - `https://operatti.com/` - Optymalizacja rodzaju pojazdu do zapotrzebowania i budżetu przeznaczonego przez władze.



Rysunek 1: Porównanie naszego rozwiązania względem innych najbardziej popularnych środków komunikacji w naszym mieście.

1.6 Repozytorium

`https://github.com/RARgames/AASD/`

1.7 Koncepcja systemu

Nasz projekt składać się będzie z symulacji, która będzie mogła stanowić PoC (Proof of Concept) dla wdrożenia naszego rozwiązania. Koncepcja naszego systemu zakłada przydzielenie czterech ról agentom: kierowcy, nadzorcy, pasażera oraz matematyka. Kierowca będzie potrzebował zdefiniowanej liczby pasażerów, których będzie mógł wziąć na raz do pojazdu. Kierowcy będą poruszali się drogami wyłącznie do zdefiniowanych punktów na mapie oznaczonych jako przystanki. Trasa pojazdu kierowcy może być dynamicznie zmieniana w trakcie jazdy. Drugą rolą dla agentów będzie rola pasażera. Pasażer będzie mógł zamówić przejazd między dwoma wskazanymi przez siebie punktami u kierowców. Pasażer następnie otrzyma możliwe opcje transportu w rozsądnym czasie. Pasażer będzie mógł zamówić przejazd na teraz oraz na wskazaną godzinę. Łącznikiem między agentami będzie nadzorca, która będzie przekazywał trasę dla kierowców oraz listę propozycji z czasami odjazdu i dojazdu dla pasażerów. Matematyk będzie odpowiedzialny za kalkulację tras. W ramach projektu na przedmiot AASD chcielibyśmy przeprowadzić symulację życia miasteczka. W symulacji planujemy porównać działanie naszego rozwiązania oraz domyślnego rozwiązania zakładającego stałą tablicę odjazdów. Miarami, które chcemy wykorzystać będzie liczba wykorzystanych pojazdów osobowych do transportu, liczba pasażerów na pojazd w mieście, potencjalne ograniczenie zużycia CO2 na obywatela, średni czas oczekiwania na przyjazd pojazdu oraz średni czas podróży użytkowników.

Symulacja pozwoli na zweryfikowanie proponowanego rozwiązania i umożliwi wdrożenie pełnego systemu (nie będzie to realizowane podczas zajęć) składającego się m.in. z podłączenia pierwszych pojazdów poprzez aplikację mobilną, którą będzie posiadał kierowca wraz z możliwymi rozszerzeniami, które będą przydatne dla automatyzacji np. pobierania płatności za przewóz większym pojazdem (autobusem) takimi jak skanery kodów QR itp. Aplikacja będzie miała dwie główne funkcje, z czego tylko jedna z nich może być aktywna w danej chwili: pasażer, przewoźnik. Kolejnym etapem, po podłączeniu pierwszych pojazdów i testowych pasażerów w systemie, będzie wstępna weryfikacja rozwiązania, która pozwoli na sprawdzenie czy mechanizmy działają poprawnie. Finalnym etapem będzie wprowadzenie rozwiązania na rynek i testy w szerszej grupie użytkowników przy jednoczesnym zbieraniu danych z działania systemu, w celu ciągłej weryfikacji poprawności działania i wprowadzania usprawnień.

2 Etap B

2.1 Wymagania systemu

1. Możliwie najwyższe wykorzystanie miejsc dostępnych w pojazdach przy jednoczesnym zapewnieniu możliwie najkrótszego czasu przejazdu i czasu oczekiwania na niego.
2. Dostosowywanie trasy kierowców do aktualnych potrzeb pasażerów (dynamiczna zmiana trasy w sytuacji dodania nowego pasażera w miejscu innym niż wcześniej zaplanowana trasa)
3. Wykorzystanie istniejącej infrastruktury przystanków.
4. Pomijanie przystanków, na których nie ma spodziewanych pasażerów tj. wybieranie możliwie najlepszej trasy dla pojazdów.
5. Możliwość zamówienia przejazdu między dwoma wskazanymi punktami.
6. Możliwość wyboru przystanku początkowego przez pasażera przy jednoczesnym proponowaniu możliwie optymalnego przystanku przez system. W przypadku wyboru innego przystanku koszt przejazdu może być wyższy.
7. Możliwość wskazania przez pasażera preferowanej godziny odjazdu lub wybrania opcji przejazdu „na teraz”.
8. Możliwość zamówienia przejazdu przy pomocy urządzenia znajdującego się na przystanku.

2.2 Role

- Pasażer
- Kierowca
- Nadzorca
- Matematyk

2.2.1 Wymagania dla roli Nadzorcy:

1. Zapewnia możliwie najlepsze trasy przejazdu dla kierowców.
2. Zapewnia możliwie najlepszych kierowców/pojazdy dla pasażerów.

2.2.2 Wymagania dla roli Matematyka:

1. Obliczenie i przedstawienie możliwie najlepsze trasy przejazdu dla kierowców.
2. Obliczenie i przedstawienie możliwie najlepszych kierowców/pojazdy dla pasażerów.

2.2.3 Wymagania dla roli Kierowcy:

1. Posiada zdefiniowaną, maksymalną liczbę pasażerów, których może zabrać do pojazdu.
2. Porusza się wyłącznie między zdefiniowanymi punktami na mapie (przystanki autobusowe).
3. Trasa przejazdu kierowcy może być dynamicznie zmieniana w zależności od zapełnienia pojazdu i zapotrzebowania na przejazdy.

2.2.4 Wymagania dla roli Pasażera:

1. Ma możliwość zamówienia przejazdu między dwoma wskazanymi punktami
2. Ma możliwość wskazania preferowanej godziny odjazdu lub wybrania opcji przejazdu „na teraz”
3. Otrzymuje opcje transportu, które zapewnią mu dotarcie w docelowe miejsce w rozsądnym czasie

2.3 Komunikacja między rolami

Przypadki użycia:

- Wyszukanie opcji przejazdu
    1. żądanie przejazdu przekazane do nadzorcy: Pasażer-Nadzorca
    2. komunikacja z potencjalnymi kierowcami (zebranie istotnych danych takich jak położenie, liczba wolnych miejsc, stan baku (ile może przejechać))
    3. potencjalna komunikacja z systemem obliczeniowym (aktualnie komunikacja Nadzorca-Matematyk)
- Wybranie przejazdu
    1. Przekazanie użytkownikowi proponowanych autobusów (Nadzorca-Pasażer)
    2. Akceptacja przejazdu przez pasażera (Pasażer-Nadzorca)
    3. Ewentualna zmiana trasy autobusu i rezerwacja miejsca (Nadzorca-Kierowca)
    4. Potwierdzenie przejazdu (Nadzorca-Kierowca)
- Wsiadanie do pojazdu
    1. Zeskanowanie kodu QR (aplikacji/karty) wewnątrz autobusu - wysłanie komunikatu do Kierowcy (Pasażer-Kierowca)
    2. Odebranie przez Kierowcę komunikatu i walidacja Pasażera z listą oczekiwanych pasażerów
    3. Wysłanie do Pasażera komunikatu o akceptacji/odrzuceniu go (Kierowca-Pasażer)

4. Zwiększenie liczby zajętych miejsc
5. Wysłanie do Nadzorcy informacji o zajściu (Kierowca-Nadzorca)

- Wysiadanie z pojazdu

1. Zeskanowanie kodu QR (aplikacji/karty) wewnątrz autobusu - wysłanie komunikatu do Kierowcy (Pasażer-Kierowca)
2. Odebranie przez Kierowcę komunikatu i walidacja czy Pasażer nie wysiada na przystanku dalszym niż zamierzał
3. Jeśli przystanek jest dalszy:
   - Wysłanie do Nadzorcy informacji o zdarzeniu (Kierowca-Nadzorca)
   - Wysłanie przez Nadzorcy informacji do Pasażera o dodatkowym obciążeniu konta (Nadzorca-Pasażer)
4. Zmniejszenie liczby zajętych miejsc
5. Wysłanie do Nadzorcy informacji o zajściu (Kierowca-Nadzorca)

- Zgłoszenie awaryjnego zdarzenia (Kierowca-Nadzorca)
- Przekazanie (aktualizacja) trasy przejazdu (w tym wysłanie kierowcy na pętle w przypadku braku pasażerów) (Nadzorca-Kierowca)

Podział komunikatów ze względu na uczestniczące podmioty:

- Matematyk-Nadzorca

1. Zapytanie o obliczenie trasy (N->M)
2. Powiadomienie o obliczonej trasie (M->N)

- Kierowca-Nadzorca

1. Powiadomienie o zdarzeniu awaryjnym (K <-> N)
2. Powiadomienie o zmianie trasy (N->K)
3. Zapytanie o stan pojazdu (N->K)
4. Powiadomienie o stanie pojazdu (K->N)
5. Powiadomienie o wejściu do pojazdu przez pasażera (K->N)
6. Powiadomienie o wyjściu z pojazdu przez pasażera (K->N)

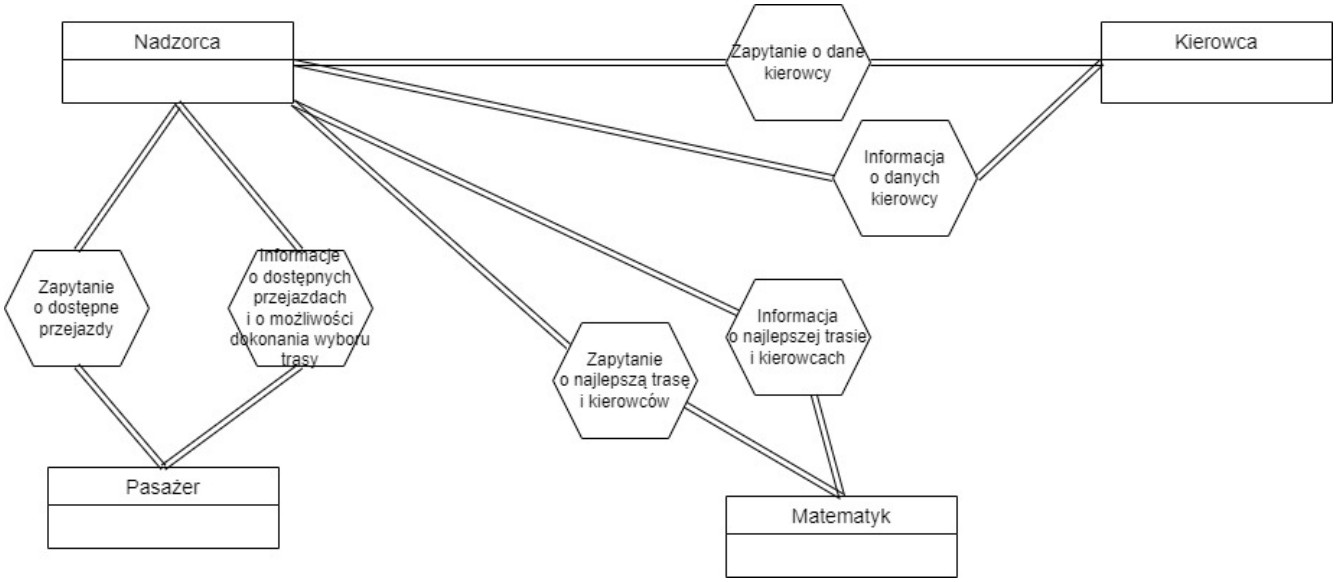
- Pasażer-Nadzorca

1. Zapytanie o przejazd (P->N)
2. Powiadomienie o dostępnych przejazdach (N->K)
3. Zamówienie przejazdu (P->N)
4. Potwierdzenie przejazdu (N->P)
5. Anulowanie przejazdu (P->N)

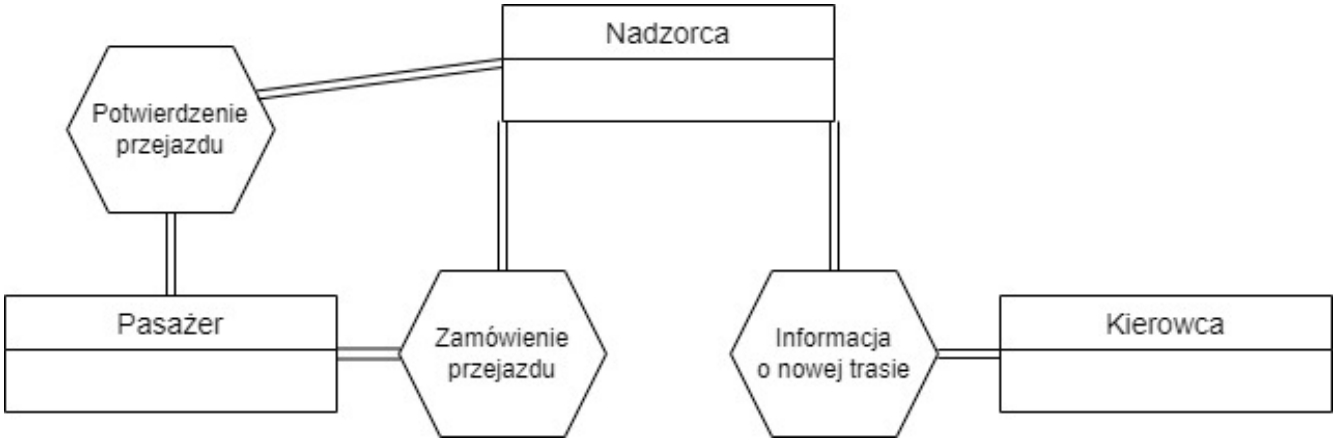
- Kierowca-Pasażer

1. Powiadomienie o wejściu do pojazdu (P->K)
2. Powiadomienie o wyjściu z pojazdu (P->K)

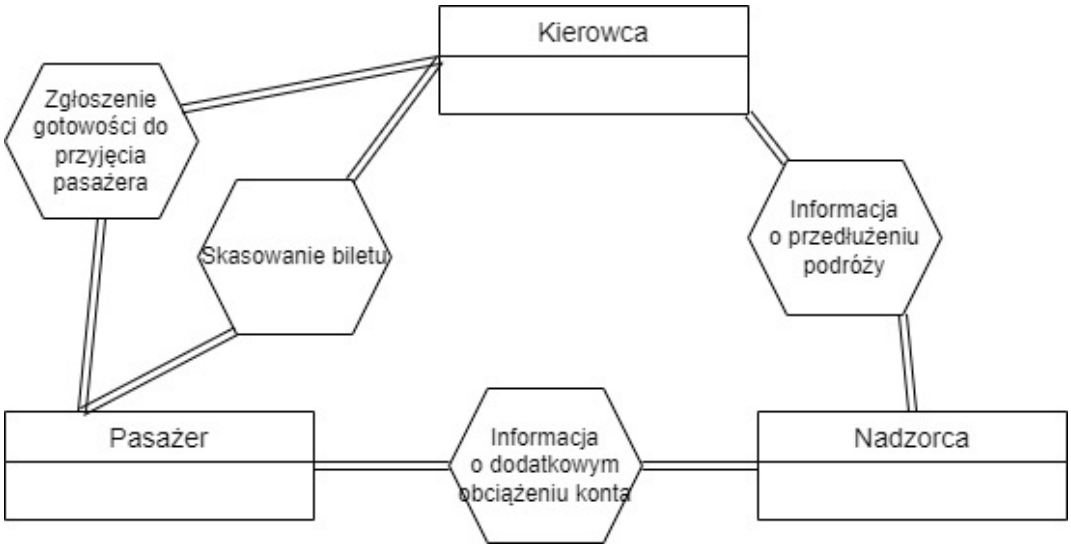
2.4 Diagramy konwersacji



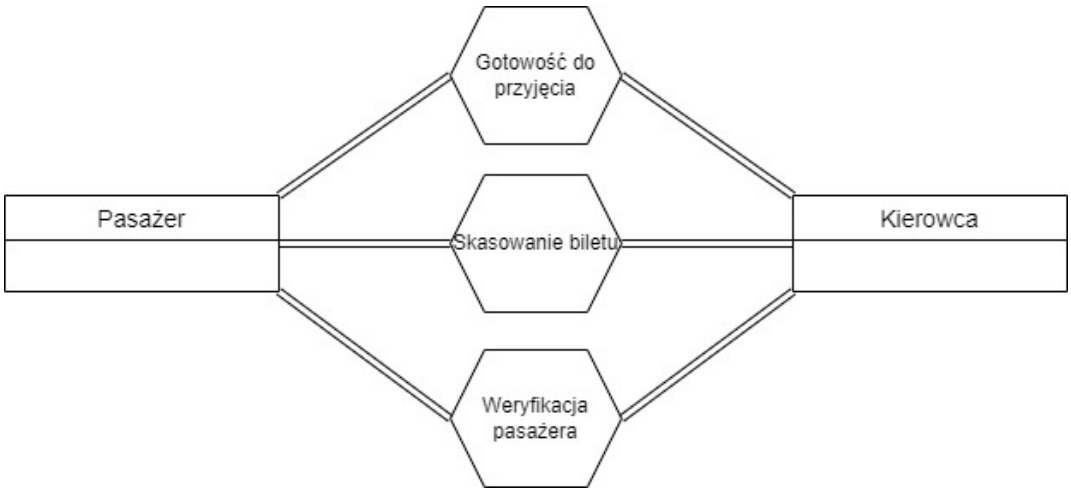
Rysunek 2: Diagram konwersacji dla wyszukiwania opcji przejazdu.



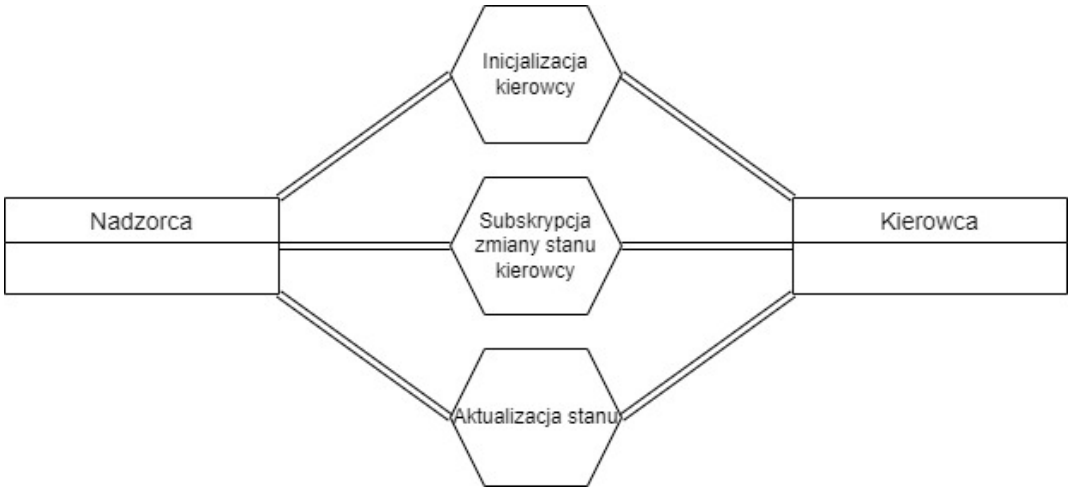
Rysunek 3: Diagram konwersacji dla wybrania przejazdu.



Rysunek 4: Diagram konwersacji dla wysiadania z pojazdu.

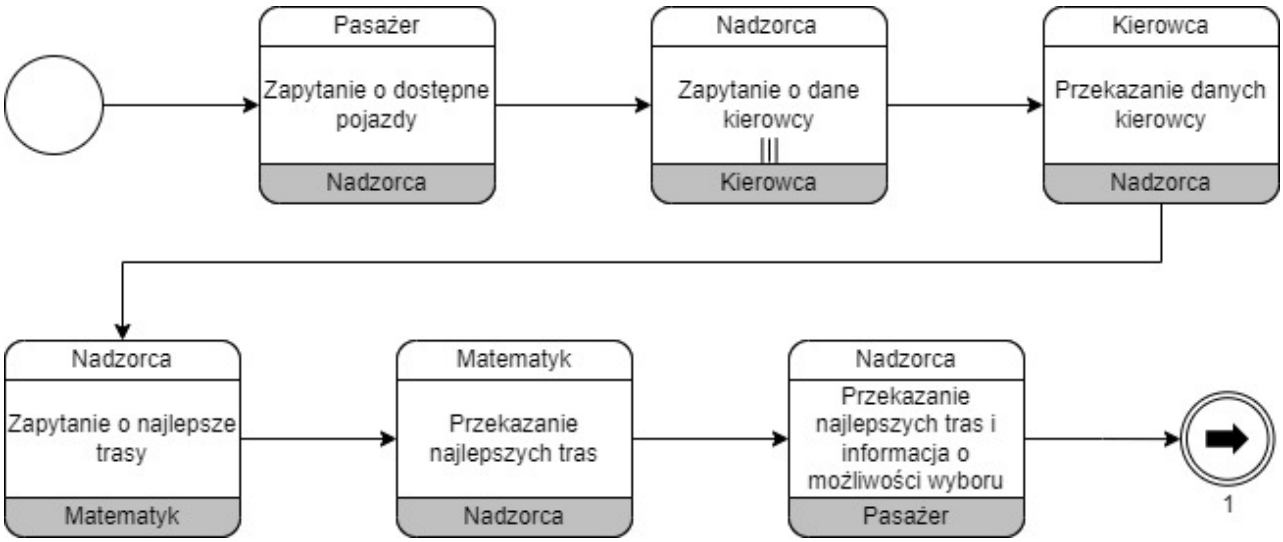


Rysunek 5: Diagram konwersacji dla wsiadania do pojazdu.

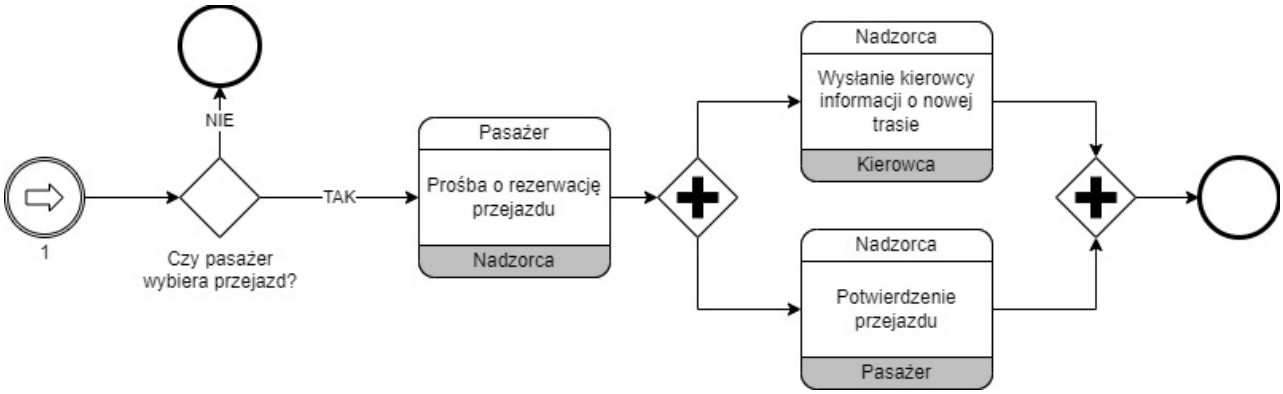


Rysunek 6: Diagram konwersacji dla subskrypcji stanu kierowcy.

2.5 Diagramy choreografii

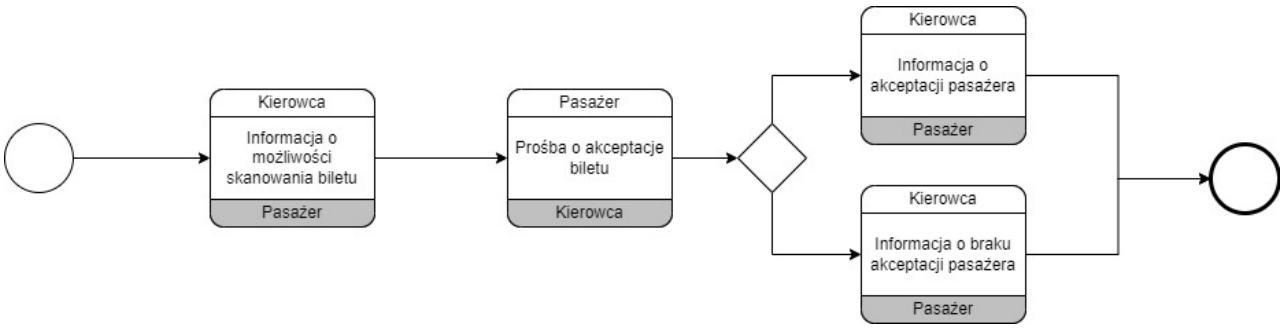


Rysunek 7: Diagram choreografii dla wyszukiwania opcji przejazdu.

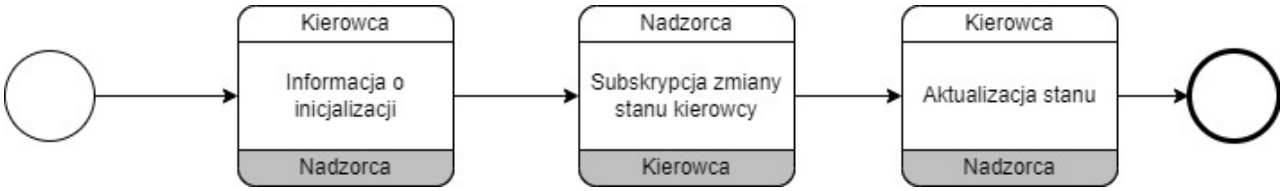


Rysunek 8: Diagram choreografii dla wybrania przejazdu.

Rysunek 9: Diagram choreografii dla wysiadania z pojazdu.

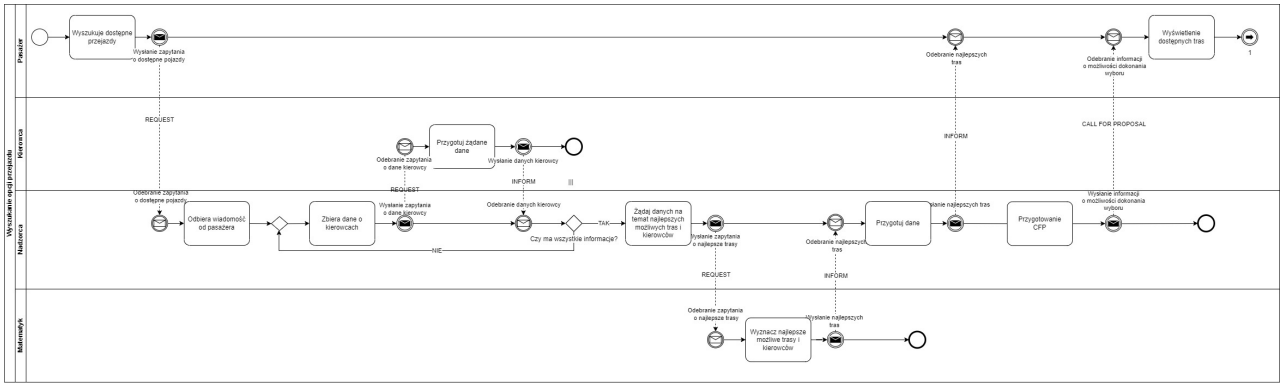


Rysunek 10: Diagram choreografii dla wsiadania do pojazdu.

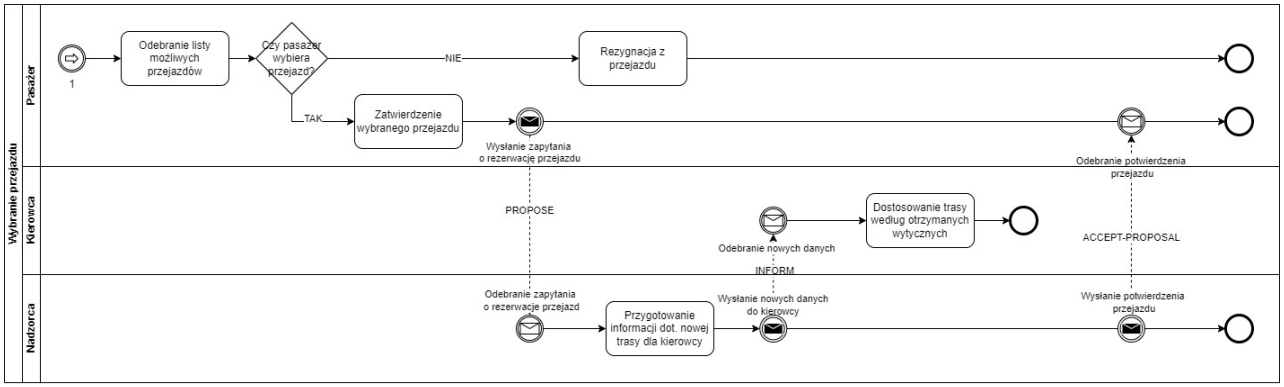


Rysunek 11: Diagram choreografii dla subskrypcji stanu kierowcy.

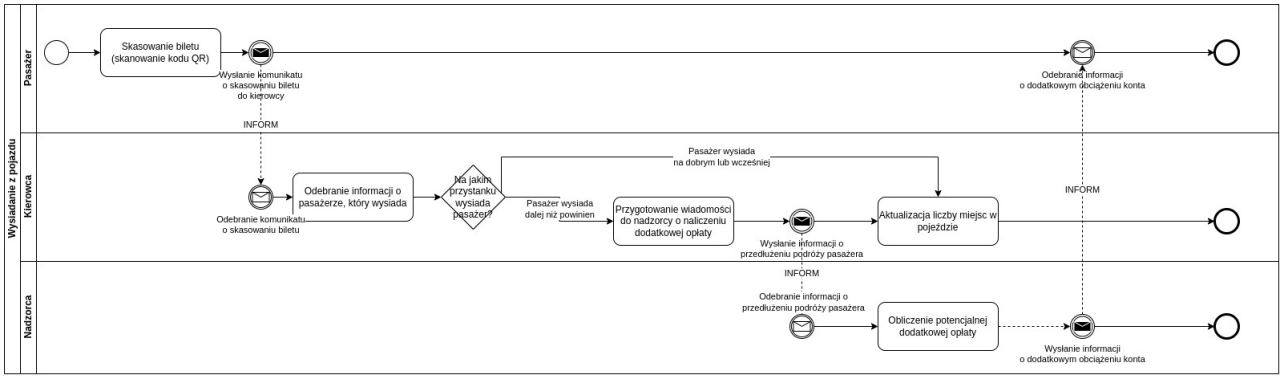
2.6 Diagramy kolaboracji



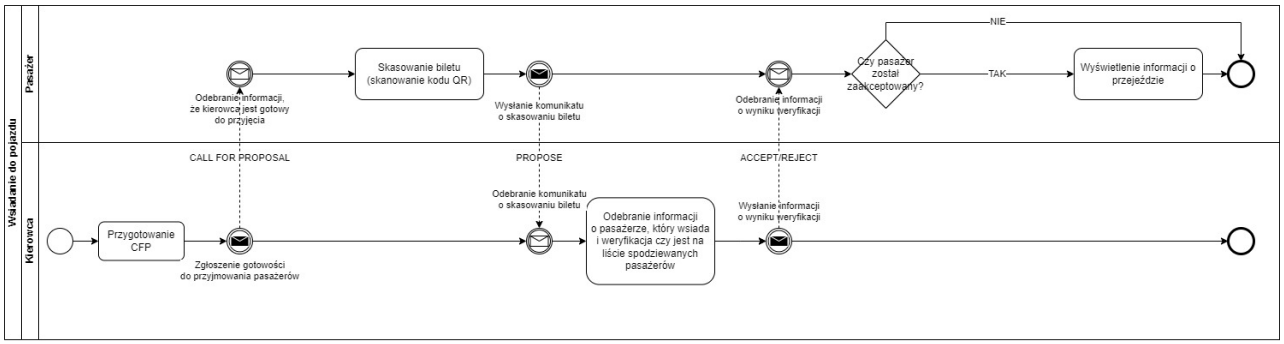
Rysunek 12: Diagram kolaboracji dla wyszukiwania opcji przejazdu.



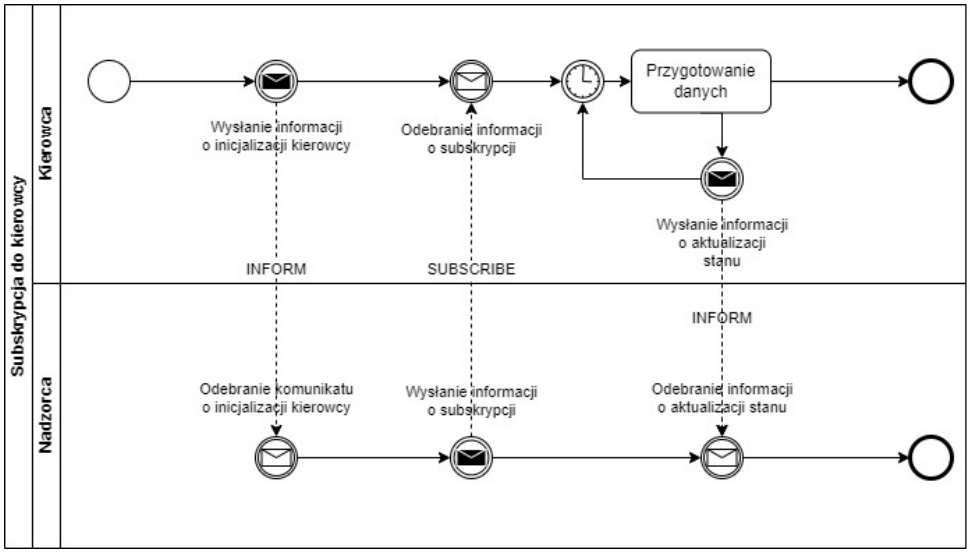
Rysunek 13: Diagram kolaboracji dla wybrania przejazdu.



Rysunek 14: Diagram kolaboracji dla wysiadania z pojazdu.



Rysunek 15: Diagram kolaboracji dla wsiadania do pojazdu.



Rysunek 16: Diagram kolaboracji dla subskrypcji stanu kierowcy.

3 Etap C i D

Przed podjęciem się realizacji projektu ustaliliśmy wraz z Opiekunem, że zaczniemy od zaimplementowania najważniejszych funkcjonalności, tj. wyszukiwania przejazdu i wybrania go (przedstawionych na dwóch pierwszych diagramach kolaboracji - 12, 13). Decyzja była podyktowana tym, że oprócz faktu, iż bez tych dwóch funkcjonalności system nie ma sensu, to wykorzystują one wszystkich zaproponowanych agentów i są wystarczające do ukazania możliwości systemu wieloagentowego.

Jako kolejny etap implementacji przyjęliśmy wsiadanie oraz wysiadanie z pojazdu, ale z pominięciem całego modułu płatności za przejazd. Moduł ten wraz ze zbieraniem statystyk przez Nadzorcę, miałby zostać dodany dopiero w końcowej fazie projektu.

Do zaprogramowania systemu wybraliśmy poznany na wykładzie python-owy framework SPADE.

Struktura wiadomości FIPA w naszym projekcie:

(**performatywa** (Inform, Receive, Subscribe, Request, CallForProposal, Accept, Propose)

:sender - identyfikator agenta nadawcy

:to - identyfikator agenta odbiorcy

:language - język treści wiadomości, który przyjmuje dla naszego projektu wartość "JSON"

:ontology - ontologia, w naszym projekcie "DoCeluMainOntology"

:body - treść wiadomości zakodowana w języku z atrybutu :language

:behaviour - identyfikator zachowania agenta (dodatkowe pole zdefiniowane na potrzeby implementacji z wykorzystaniem zachowań)

)

3.1 Sposób implementacji

Kod programu został podzielony na kilka pakietów:

- agents - pakiet zawierający klasy poszczególnych agentów
- entities - pakiet z klasami wykorzystywanymi do przesyłania informacji między agentami
- behaviours, messages - pakiety z klasami nadpisującymi domyślne implementacje zachowań, wiadomości i szablonów wiadomości
- utils - pakiet ze zmiennymi statycznymi i funkcjami wykorzystywanymi w całym projekcie

3.1.1 Implementacja agenta

Nasz agent został zaimplementowany przy wykorzystaniu klasy *Agent* z biblioteki Python-owej SPADE. W klasie każdego agenta zaimplementowaliśmy klasy przedstawiające zachowania dziedziczące po klasie *BaseOneShotBehaviour*, *BasePeriodicBehaviour* lub *BaseCyclicBehaviour*, które zostały przez nas przygotowane w pakiecie *behaviours*. Wszystkie klasy zaimplementowane w ten sposób wykorzystują mechanizmy przedstawione w klasie *CyclicBehaviour* z biblioteki SPADE. Zachowania typu *OneShot* charakteryzują się tym, że po wykonaniu metody *run()* zachowanie się kończy. Zachowania typu *Cyclic* wykonują metodę *run()* do momentu wystąpienia warunku stopu. Zachowania typu *Periodic* wykonują metodę *run()* co określony okres. Do każdej klasy załączyliśmy w prywatnych zmiennych informacje o systemie logującym oraz dane pliku konfiguracyjnego. Pierwszą metodą uruchamianą po starcie inicjalizacji agenta jest *setup*, w metodzie tej wykonywana jest logika odpowiedzialna za poprawne przygotowane agenta do „pracy".

Przykład implementacji agenta kierowcy wraz z zachowaniami został przedstawiony poniżej. W poniższym fragmencie kodu przedstawiono istotne fragmenty implementacji agenta tj. parametry klasy, w których znajdują się deklaracje zachowań i zmiennych reprezentujących stan agenta, konstruktor oraz metodę *setup*. W konstruktorze poza wywołaniem konstruktora klasy bazowej, ustawiany jest początkowy stan agenta oraz obiekty z konfiguracją i funkcjonalnością logowania. Opis implementacji zachowań w klasie agenta został pominięty, natomiast w kolejnym fragmencie kodu przedstawiono implementację jednego wybranego zachowania. W metodzie *setup* inicjalizowane są zachowania i ewentualnie korespondujące im wiadomości z szablonami.

```
class DriverAgent(agent.Agent):
    # Behaviours
    inform_driver_data: 'InformDriverData'
    subscribe_to_manager: 'SubscribeToManager'
    receive_request_driver_data: 'ReceiveRequestDriverData'
    receive_request_driver_data_template: DriverDataTemplate
    receive_inform_path_change: 'ReceiveInformPathChange'
    receive_inform_path_change_template: PathChangeTemplate
    # Agent state
    __capacity: int
    __current_path: Optional[Any] = None
    __geolocation: Dict[str, Any]

    _logger: Logger
    _config: Config

    def __init__(self, jid: str, password: str, capacity: int,
                 geolocation: Dict[str, Any], verify_security: bool = False):
        super().__init__(jid, password, verify_security=verify_security)
        self._config = get_config()
        self._logger = get_logger(LOGGER_NAME)
        self.__capacity = capacity
        self.__geolocation = geolocation

    class InformDriverData(BaseOneShotBehaviour):
        # behaviour implementation ...

    class ReceiveRequestDriverData(BaseCyclicBehaviour):
        # behaviour implementation ..

    class ReceiveInformPathChange(BaseCyclicBehaviour):
        # behaviour implementation ...

    class SubscribeToManager(BasePeriodicBehaviour):
        # behaviour implementation ...

    async def setup(self):
        self._logger.info('DriverAgent started')
        await self._setup_receive_request_driver_data()
        await self._setup_receive_inform_path_change()
        await self._setup_inform_driver_data()
        await self._setup_subscribe_to_manager()
        self.add_behaviour(self.receive_request_driver_data,
        ↪  self.receive_request_driver_data_template)
        self.add_behaviour(self.receive_inform_path_change,
        ↪  self.receive_inform_path_change_template)
        self.add_behaviour(self.subscribe_to_manager)
```

Poniżej przedstawiliśmy naszą implementację przykładowego zachowania agenta, gdzie pokazujemy strukturę klasy zachowania, cztery podstawowe metody, które można nadpisać względem domyślnej implementacji zaproponowanej w bibliotece SPADE. W metodzie *on_start()* definiowane są komendy, które są wywoływane rutynowo przed każdym uruchomieniem głównej pętli zachowania. W metodzie *on_end()* definiujemy komendy wykonywane po zakończeniu zachowania lub gdy zachowanie jest zabijane. W naszym projekcie podstawowymi komendami, które dodawaliśmy do tych metod są komendy logujące rozpoczęcie i zakończenie działania zachowania. Najważniejszą metodą w klasie jest metoda *run()* definiująca co dokładnie zachowanie będzie robić. W tej metodzie deklarowaliśmy też warunki stopu.

```
class ReceiveRequestDriverData(BaseCyclicBehaviour):
    agent: 'DriverAgent'

    def __init__(self,):
        super().__init__(LOGGER_NAME)

    async def on_start(self):
        self._logger.info('ReceiveRequestDriverData running...')

    async def run(self):
        msg = await self.receive()
        if msg:
            self._logger.debug(f'Message received with content: {msg.body}')
            if not self.agent.has_behaviour(self.agent.inform_driver_data):
                self._logger.debug('InformDriverData renewed')
                await self.agent._setup_inform_driver_data()
                self.agent.add_behaviour(self.agent.inform_driver_data)

    async def on_end(self):
        self._logger.info('ReceiveRequestDriverData ending...')
```

3.2 Sposób komunikacji

3.2.1 Użyte performatywy

Performatywy, które wykorzystaliśmy w implementacji są zgodne z zaproponowanymi na etapie projektowania systemu w sekcji diagramów kolaboracji 2.6.

3.3 Napotkane problemy

1. Do uruchomienia systemu wymagane było „podpięcie" serwera xmpp[1].

   W związku z tym, że nie chcieliśmy rejestrować go w sieci, zdecydowaliśmy się wykorzystać obraz dockerowy *ejabberd/ecs*[2]. Konfiguracja oraz skrypty dodające nowych agentów znajdują się w folderze *containers* w repozytorium projektu.

2. Jednym z napotkanych problemów było pobieranie stanu pojazdów przez zarządcę systemu w sytuacji, gdy potrzebne było wyliczenie najlepszych tras dla danego żądania.

   Zgodnie z projektem systemu, problem ten nie powinien wystąpić, gdyż obserwowanie stanu pojazdów miało odbyć się dzięki subskrypcji, jednak początkowo nie planowaliśmy tej części realizować. Ostatecznie, przy inicjalizacji agenta Kierowcy, wysyłany jest komunikat do Nadzorcy, który to następnie dodaje danego agenta do swojej sieci kontaktów (w silniku SPADE można traktować to jako subskrypcję).

3. Problemem okazał się sam model matematyczny, który optymalizowałby trasy dla wielu kierowców.

   W aktualnej wersji systemu trasy nie są optymalizowane na bieżąco, a jedynie „wrzucane" na koniec. Z dostępnych przejazdów wybierany jest ten, któremu najkrócej zajmie, biorąc jego aktualne obłożenie, dojechanie do klienta i dowiezienie go do celu.

4. Brak możliwości usunięcia agenta z listy kontaktów drugiego agenta.

   Taki rezultat powinien być osiągany przez usunięcie subskrypcji, przynajmniej według naszego rozumowania (w dokumentacji nie ma o tym informacji), jednakże nie udało się doprowadzić ów funkcjonalności do działania. Działająca funkcjonalność byłaby wygodna w kontekście rosnącej listy kontaktów nadzorcy w przypadku napływających agentów klientów.

[1] https://xmpp.org/about/technology-overview/
[2] https://www.ejabberd.im/

3.4 Opis algorytmów

3.4.1 Wyliczanie tras

Głównym zadaniem agenta matematyka jest w naszym projekcie wyliczanie najtańszych tras pod względem kosztów czasu przejazdu. W tym celu napisaliśmy algorytm, który wylicza najbardziej korzystną trasę dla klienta. Algorytm pobiera graf z zapisanymi odległościami pomiędzy przystankami, aktualne trasy autobusów, oraz przystanki początkowy i końcowy klienta, by ocenić jak długo będzie czekał na autobus i zdecydować, który autobus będzie dla niego najbardziej korzystny. Poniżej prezentujemy pseudokod algorytmu wyliczającego trasę dla autobusu.

```
wczytaj(grafkosztów) - macierz
wczytaj(trasy) - 2 wymiarowa lista
wczytaj(listazapotrzebowania) - 2 wymiarowa tablica
l.autobusów ← długość trasy
Dla i ← 0 do długości listazapotrzebowania powtarzaj:
    temp ← 0
    minkoszt ← 9999
    Dla j ← 0 do l.autobusów powtarzaj:
        trasa ← trasy[j]
        koszt ← 0
        Jeżeli długość trasy[j] == 1 to
            tymczasowa ← j
            break
        W przeciwnym razie
            tymczasowa2 ← zapotrzebowanie[i]
            Dla k ← 0 do długość trasy powtarzaj:
                koszt ← koszt + grafkosztów[trasa[k]][trasa[k+1]]
            Jeżeli tymczasowa2[0] == l.autobusów[długość trasa -1] to
                koszt ← koszt + grafkosztów[długość
                ↪  trasa-1][listazapotrzebowania[i][1]]
            W przeciwnym razie
                koszt ← koszt + grafkosztów[długość
                ↪  trasa-1][listazapotrzebowania[i][1]] +
                ↪  grafkosztów[[listazapotrzebowania[i][1]][listazapotrzebowania[i][0]]
            Jeżeli koszt < minkoszt
                minkoszt ← koszt
                tymczasowa ← j
    trasa ← trasy[tymczasowa]
    tymczasowa2 ← zapotrzebowanie[i]
    Jeżeli trasa[długość trasy -1] != tymczasowa2[0]
        trasa[długość trasa + 1] = tymczasowa2[0]
    trasa[długość trasa + 1] = tymczasowa2[1]
    trasy[tymczasowa] = trasa
zwróć trasy
```

3.5 Testy aplikacji

3.5.1 Testy jednostkowe agentów

Dla wybranych agentów przygotowane zostały testy jednostkowe, w których sprawdzane są m.in. następujące rzeczy:

- poprawność dopasowania wiadomości do szablonów,
- poprawność wykonania (i zakończenia) zachowań,
- poprawność zmian stanów.

3.5.2 Testy integracyjne

Test integracyjny weryfikuje poprawność dwóch przypadków użycia - wyszukanie oraz wybór opcji przejazdu.

W folderze *examples* powstał plik *manager_ driver_ communication.py* wspomagający testy integracyjne komunikacji między agentami Nadzorcą i Kierowcą. Skrypt ten uruchamia jedną instancję Nadzorcy oraz trzech agentów - Kierowców, a następnie wywołuje konkretne zachowania Nadzorcy, na które później odpowiadają Kierowcy (jako reakcja na otrzymanie komunikatu).

Weryfikacja poprawności odbywała się manualnie, tj. uruchamiający sprawdzał poprawność logów wyświetlających się w konsoli.

3.6 Opis braków systemu

- W systemie nie zostały zaimplementowane inne przypadki użycia, oprócz wyszukania i wybrania opcji przejazdu - zgodnie z założeniami przed rozpoczęciem procesu implementacji (odn. 3)
- Wyszukiwanie przyszłych połączeń - nie udało się zaimplementować algorytmu agenta Matematyka w wersji, która uwzględniałaby inne daty początku podróży niż data, w której jest wysyłane żądanie. W związku z tym, nie uwzględniliśmy tego też w żądaniach między agentami Klient - Nadzorca.

Literatura

[1] Wykluczeni. o likwidacji transportu zbiorowego na wsi i w małych miastach, 2019.

[2] Wykluczenie transportowe mieszkańców mniejszych miejscowości, 2018.

[3] Brak dofinansowania wśród powodów wykluczenia transportowego, 2019.