

# AKADEMIA GÓRNICZO-HUTNICZA

Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki



## DISPATCH RIDER

**HOLONICZNY SYSTEM DO ROZWIĄZYWANIA DYNAMICZNEGO  
PROBLEMU TRANSPORTOWEGO**

**DOKUMENTACJA PROJEKTOWA**

Kraków, sierpień 2012

## 1. Spis treści

2.	Wstęp .....	5
2.1.	Problematyka zagadnienia .....	5
2.2.	Wprowadzenie do problemu PDPTW .....	5
2.3.	Cele i zadania systemu.....	8
2.4.	Analiza wymagań .....	9
2.5.	Koncepcja rozwiązania z uwzględnieniem modelu systemu agentowego .....	10
2.5.1.	Agenci w systemie .....	10
2.5.2.	Współpraca agentów .....	14
3.	Działanie systemu.....	15
3.1.	Przypadki użycia.....	15
3.2.	Agenci w systemie – diagramy .....	18
3.3.	Poglądowy opis symulacji .....	26
3.3.1.	Krótki opis architektury systemu.....	28
3.4.	Tworzenie ofert do powstania nowego holonu.....	30
3.4.1.	Procedura negocjacji .....	30
3.4.2.	Wymiana elementów .....	36
3.5.	Realizacja wysyłania zleceń paczkami .....	37
3.5.1.	Przebieg tworzenia holonów w podejściu, gdzie zlecenia są wysyłane paczkami	37
3.5.2.	Algorytm tworzenia listy preferencji w przypadku wysyłania paczkami .....	38
3.5.3.	Częste problemy .....	39
3.6.	Funkcje kosztu .....	40
3.7.	Tworzenie jednostek transportowych – Eunity .....	40
3.8.	Rozdział zleceń przez Dystrybutora .....	41
3.8.1.	Preferowany koszt .....	41
3.8.2.	Preferowane wykorzystanie .....	41
3.9.	Zaimplementowane algorytmy .....	41
3.9.1.	Algorytmy wstawiania zleceń .....	42
3.9.2.	Dodatkowe klasy i metody .....	45
3.9.3.	Simulated trading.....	46
3.8.4.	Wzorzec sytuacyjny.....	57
1.9.	Miary.....	60
3.9.1.	Wyliczanie miar.....	60
3.9.2.	Zapisywanie miar .....	61
3.9.3.	Zaimplementowane kalkulatory miar .....	62
3.9.4.	Wizualizacja miar .....	70

3.9.5.	Diagramy .....	71
3.10.	Dynamiczna zmiana konfiguracji.....	73
3.11.	Miękkie okna czasowe .....	74
3.11.1.	Funkcja kary .....	74
3.11.2.	Wartości domyślne – wagi zleceń .....	75
3.11.3.	Zmiany w implementacji algorytmów przydziału zleceń i ST .....	76
3.11.4.	Sposób działania .....	76
3.12.	Optymalizacja algorytmu complexSimulatedTrading .....	77
3.13.	Uczenie maszynowe .....	77
3.13.1.	QLearning .....	78
3.13.2.	Stany .....	80
3.13.3.	Klastrowanie .....	84
3.14.	Graf.....	91
3.14.1.	Generacja grafu.....	91
3.14.2.	GraphSchedule.....	91
3.14.3.	Wyszukiwanie tras w grafie.....	91
3.15.	Niepewne czasy przejazdu .....	92
3.15.1.	Realizacja.....	92
3.15.2.	Opóźnienia(zaburzenia) w grafie.....	93
3.15.3.	Różne czasy dowiadywania się o zmianach w grafie .....	94
3.15.4.	Algorytmy predykcji zmian w grafie.....	95
3.16.	Używanie agentów pomocniczych.....	95
3.16.1.	Tworzenie agentów pomocniczych .....	96
3.16.2.	Konfigurowanie agentów pomocniczych .....	96
3.16.3.	Używanie agentów pomocniczych .....	96
3.17.	Komparatory zleceń przyznawanych przez Dystrybutora .....	97
3.17.1.	Tworzenie nowych komparatorów .....	97
3.17.2.	Konfigurowanie użycia komparatora .....	97
3.18.	Interfejs GUI .....	97
3.18.1.	Suwak czasu.....	97
3.18.2.	Zakładki .....	98
3.18.3.	Zmiany wprowadzone w systemie .....	104
3.18.4.	Architektura GUI .....	105
4.	Opis sposobu wprowadzania niektórych zmian .....	107
4.9.	Zmiany konfiguracji – dodawanie nowych parametrów konfiguracyjnych .....	107
4.10.	Dodanie możliwości, żeby pojazd wyjeżdżał najpóźniej jak się da .....	108

4.11.	Wprowadzenie kolejkowania zleceń, które są potem przekazywane przez Dystrybutora .....	109
4.12.	Wprowadzanie algorytmów optymalizacji działających podobnie do obecnego SimulatedTrading .....	110
4.12.1.	Implementacja klasycznego Simulated Trading .....	112
4.13.	Reorganizacja holonów – wytyczne .....	112
4.14.	Mechanizm dodawania nowych algorytmów uczenia maszynowego .....	113
4.15.	Dodanie dedykowanego algorytmu dla miękkich okien .....	114
4.15.1.	Wersja z implementacją wyliczania danych w klasach algorytmów .....	114
4.15.2.	Wersja z implementacją nowej klasy Schedule .....	115

## 2. Wstęp

### 2.1. Problematyka zagadnienia

Zastosowanie skutecznych metod zarządzania transportem ma kluczowe znaczenie przy konkurencyjności firm spedycyjnych. Optymalne rozmieszczenie zasobów i przydział zadań w znacznym stopniu może wpływać na ponoszone koszty. Sprawia to, że zagadnieniu temu poświęca się coraz więcej uwagi.

Problemy transportowe zwykle polegają na przewiezieniu towaru z jednego miejsca do drugiego w określonych ramach czasowych. Problemy takie zwykle opiera się na uproszczonych modelach świata, które nie zawsze zawierają wszystkie elementy składające się na rzeczywiste rozwiązanie. Do problemów takich możemy zaliczyć VRP (*Visual Routing Problem*) oraz PDP (*Pickup and Delivery Problem*), a także ich wersje z oknami czasowymi VRPTW (*Visual Routing Problem with Time Windows*) PDPTW (*Pickup and Delivery Problem with Time Windows*). Istotnymi elementami, który nie został uwzględniony w wyżej wymienionym jest zróżnicowanie pojazdów obsługujących zlecenia oraz możliwość składania pojazdów z elementów (np. ciągnik i naczepa). Celem pracy jest realizacja systemu opartego na architekturze holonicznej. Zastosowanie tego podejścia pozwala na rozbicie problemu na szereg podproblemów. Ponadto system holoniczny pozwoli nam na budowanie zespołów transportowych ze zróżnicowanych elementów.

### 2.2. Wprowadzenie do problemu PDPTW

Problem PDPTW (*Pickup and Delivery Problem with Time Windows*) jest uogólnieniem problemu VRPTW (*Visual Routing Problem with Time Windows*), który jest z kolei uogólnieniem problemu komiwojażera. Oznacza to, że jest to problem NP-zupełny.

Rozwiązywanie problemu PDPTW polega na znalezieniu zbioru tras przejazdu dla floty przejazdów w celu wykonania zestawu zleceń transportowych. Każdy pojazd ma określoną ładowność. Każde zlecenie jest zdefiniowane przez punkty załadunku i wyładunku, wymagana ładowność oraz przez czasy odbioru i dostarczenia zleceń. Dodatkowo zdefiniowany jest punkt początkowy i końcowy dla każdej z tras.

Problem PDPTW może posiadać szereg ograniczeń. Definiują one zasady jakimi będziemy się kierować przy znajdowaniu rozwiązania. Zaliczają się do nich

ograniczenie pierwszeństwa, ładowności, par i zasobów. Ograniczenie pierwszeństwa oznacza, że miejsca załadunku muszą zostać odwiedzone przed miejscami rozładunku. Ograniczenie to nie jest opcjonalne i musi być spełnione w każdym przypadku. Ograniczenie ładowności oznacza ładowność poszczególnych pojazdów. Żaden pojazd nie może przewieźć więcej niż wynosi jego ładowność. Ograniczenie par oznacza nierozdzielnosć zleceń, każde zlecenie musi zostać wykonane przez jeden pojazd. Ograniczenie zasobów oznacza ograniczenie liczby pojazdów dostępnych przy znajdowaniu rozwiązania.

W zależności od oczekiwanej efektu końcowego możemy wyróżnić kilka celów poszukiwania rozwiązania. Istnieje kilka kryteriów według których możemy stwierdzić, że rozwiązanie jest optymalne.

Minimalizacja liczby pojazdów oznacza, że rozwiązaniem jest to, w którym liczba pojazdów będzie najmniejsza. Minimalizacja czasu to poszukiwanie rozwiązania, w którym wszystkie zlecenia zostaną wykonane w najkrótszym czasie. Minimalizacja dystansu oznacza poszukiwanie możliwie najkrótszych tras dla pojazdów. Minimalizacja czasu oczekiwania, oznacza poszukiwanie rozwiązania, w którym czas oczekiwania na realizację zleceń jest najmniejszy.

Minimalizacja złożona jest to poszukiwanie rozwiązania które spełnia powyższe warunki w dowolnej kombinacji, dla każdego warunku przypisując określoną wagę.

Na potrzeby projektu problem PDPTW zostanie rozszerzony o poniższe punkty. Celem rozszerzenia jest dostosowanie problemu do bardziej realistycznych warunków.

#### **a) Sieć komunikacyjna**

W problemie PDPTW ma postać grafu pełnego, rozpiętego pomiędzy punktami załadunku i wyładunku. W zastosowanym przez nas rozszerzeniu graf będzie grafem spójnym nieskierowanym, jednak nie będzie pełny.

#### **b) Zespoły transportowe**

Każde zlecenie musi być zrealizowane przez zespół transportowy składający się z trzech elementów: kierowcy, naczepy oraz ciągnika. Każdy z elementów

może posiadać szczególne dla siebie cechy, takie jak np. rodzaj przewożonego towaru lub ładowność. Cechy te mogą się różnić wartościami pomiędzy poszczególnymi elementami tego samego typu. Dzięki ich określeniu możliwe będzie lepsze dobranie zespołu do zlecenia.

**c) Dynamika systemu**

Zlecenia nie pojawią się w systemie na początku. Każde z nich będzie miało zadany czas pojawienia się w systemie. Pozwoli to na symulowanie ciągłej pracy firmy spedycyjnej.

**d) Sytuacje kryzysowe**

Są to nieprzewidziane zdarzenia mogące mieć wpływ na realizowane zlecenia. Zwykle oznaczają one ubywanie zasobów i będą wymagały zmiany przygotowanego już planu realizacji. Mogą one dotyczyć jednego z dwóch obszarów, infrastruktury drogowej lub środków transportu i ludzi.

Sytuacje związane z infrastrukturą drogową to korki i awarie dróg. Ich wpływ na rozwiązanie jest podobny. Wymagają one znalezienia innej trasy lub generują opóźnienia. Ubywanie środków transportu (awarie pojazdów) oraz zasobów ludzkich (niedyspozycja kierowcy) często powodują brak możliwości realizacji podjętych działań. Wymagają wymiany niezdolnego do działania elementu i znalezienie nowego rozwiązania.

Godnym dodania jest, iż w obecnej wersji systemu zarządzanie sytuacjami kryzysowymi zostało mocno ograniczone. Wyjątkiem jest tutaj obsługa korków.

**e) Rozproszenie zasobów.**

W systemie może występować więcej niż jedna baza. Budowanie zespołów transportowych odbywa się w założeniu podstawowym w obrębie danej bazy, jednak nie ma wymogu, by zespół wrócił do tej samej bazy po zakończeniu realizacji zleceń. Ponadto zespół może w trakcie przydzielania kolejnych zleceń zmieniać bazę do której zamierza wrócić tak, żeby koszt przejazdu był mniejszy. Rozwiązanie takie może spowodować, że niektóre małe zlecenia będą obsługiwane przez duże zespoły, ze względu na niedużą odległość punktów załadunku i rozładunku od ich bazy. W celu zwiększenia wpływu wielu baz na

wyniki przy budowaniu zespołów w ramach jednej bazy korzystne jest rozmieszczenie pojazdów w poszczególnych bazach według wielkości.

Rozszerzony aspekt budowania zespołów pozwala na składanie ich nie tylko w obrębie jednej bazy, a w wielu. Należy jednak pamiętać, że naczepa nie może podróżować bez ciągnika. Ogranicza to więc budowanie zespołów do zbierania ciągnika i kierowcy z jednej bazy, a następnie podłączenie przyczepy w innej. Do kosztu wykonania zlecenia trzeba uwzględnić koszt przejazdu niepełnego zespołu do bazy, w której znajduje się naczepa. Rozwiążanie zespołu natomiast, może nastąpić w dowolnej bazie

### 2.3. Cele i zadania systemu

Celem pracy jest stworzenie systemu funkcjonalnością zbliżonego do systemu *TeleTruck*, jednak rozwiązywającego zadania bardziej zbliżone do klasycznych, dla których istnieją dobrze opisane najlepsze znane rozwiązania. Pozwoli to na porównanie osiągniętych wyników z już istniejącymi.

Zadaniem systemu będzie rozwiązanie szeregu zadań począwszy od zadań dla statycznego problemu po dynamiczne zadania problemu rozszerzonego z zastosowaniem agentów holonicznych.

Klasyczny problem polega na znalezieniu rozwiązania zestawu zleceń transportowych, w którym cała pula zleceń jest znana w momencie rozpoczęcia obliczeń. Rozwiązanie stanowi zestaw tras zespołów transportowych. Wszystkie zespoły są takie same, a sieć komunikacyjna jest grafem pełnym. Do znalezienia tego typu rozwiązania nie wykorzystuje się elementów takich jak ciężarówki, kierowcy czy naczepy. Zespoły transportowe traktowane są jako całość. Rozwiązywanie go było jednym z celów stawianych przed systemem.

Kolejnym zadaniem było wprowadzenie dynamiki do rozwiązywanych zadań. Dynamika oznacza nieznajomość pełnej puli zleceń na początku ich realizacji. Kolejne zlecenia pojawiają się w trakcie trwania symulacji. Wymaga to od systemu dostosowanie się do ciągle zmieniającej się sytuacji. Graf sieci komunikacyjnej nie musi być grafem pełnym, jednak może nim być w celu zachowania większego podobieństwa do problemu klasycznego.

Kolejnym etapem jest rozwiązywanie zadań z użyciem holonów. Oznacza to wykorzystanie zespołów transportowych traktowanych jako osobne elementy. Problemy te mogą korzystać z niepełnego grafu sieci komunikacyjnej. Dodatkowo w celu zwiększenia przydatności holonów możliwe jest wprowadzenie więcej niż jednej bazy. Możliwe jest także zróżnicowanie elementów zespołów transportowych (np. różna ładowność naczep).

#### 2.4. Analiza wymagań

System musi pozwalać na zarządzanie przebiegiem symulacji. Oznacza to jej uruchamianie, przechodzenie do kolejnych kroków, zatrzymywanie oraz wznowianie.

System musi pozwalać na definiowanie zestawów zleceń do wykonania, wczytywanie ich z plików oraz konwersję zestawów statycznych na dynamiczne. Dla każdego zlecenia musi być możliwe zdefiniowanie jego wielkości, początkowego i końcowego okna czasowego oraz czasu nadania do systemu.

System musi pozwalać na definiowanie floty transportowej. W przypadku problemu statycznego oznacza to ograniczanie ilości zespołów transportowych. W przypadku zagadnień z holonami oznacza to możliwość definiowania ilości poszczególnych elementów, ich cech, oraz początkowego położenia (jeśli występuje więcej niż jedna baza).

Wyniki symulacji muszą być prezentowane na bieżąco w postaci komunikatów o wykonywanych aktualnie operacjach, a także jako zbiorcze informacje po zakończeniu takie jak kalendarze poszczególnych zespołów transportowych oraz zestawienie wyników (długości tras).

System musi pozwalać na generowanie testów na podstawie istniejących testów dla problemów statycznych. Musi być możliwe zamienienie problemu w problem dynamiczny z różnymi kryteriami dotyczącymi czasu nadania zleceń. Konieczne jest też umożliwienie wystąpienia więcej niż jednej bazy.

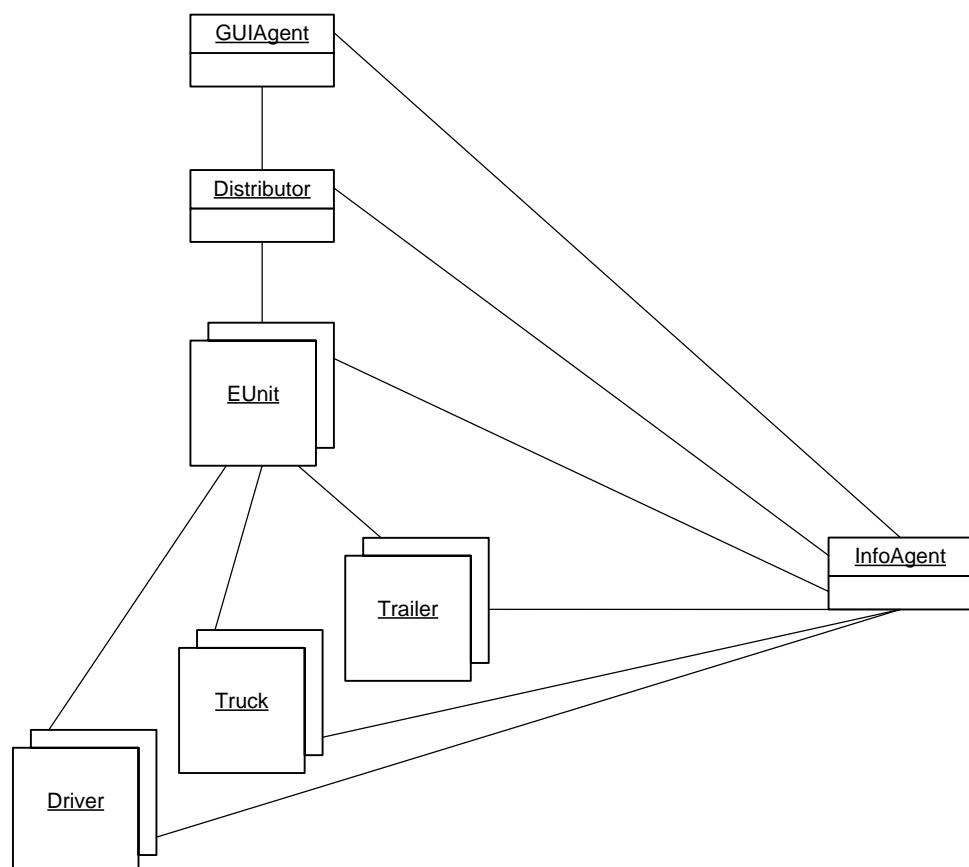
Dodatkowym zadaniem systemu jest wizualizacja przebiegu symulacji. Wizualizacja ma przedstawiać sieć komunikacyjną oraz poruszające się po niej zespoły transportowe. Musi być aktualizowana na bieżąco w trakcie trwania symulacji.

## 2.5. Koncepcja rozwiązania z uwzględnieniem modelu systemu agentowego

Projekt opiera się na platformie zarządzania agentami JADE (Java Agent DEvelopment Framework). Dokładne definicje związane z zagadnieniem agentów można znaleźć w wielu ogólnodostępnych źródłach. Dokumentacja i wszystkie niezbędne informacje na temat środowiska JADE są do uzyskania na stronie <http://jade.tilab.com>.

### 2.5.1. Agenci w systemie

System będzie posiadał pięć typów agentów odpowiedzialnych za rozwiązywanie postawionych mu problemów oraz dwa typy agentów pomocniczych. Rysunek przedstawia agentów w systemie i dostępne drogi komunikacji między nimi.



Rysunek 1 Współpraca pomiędzy agentami

Agenci odpowiedzialni za rozwiązanie problemu to: Distributor, eUnit, Driver, Trailer oraz Truck. Agentami pomocniczymi są GUIAgent oraz InfoAgent. Żaden z agentów nie jest agentem mobilnym, nie przemieszczają się oni pomiędzy platformami.

**a) Dyspozytor (*Distributor Agent*).**

Agent odpowiedzialny za przydzielanie zadań poszczególnym zespołom transportowym, przeprowadzanie aukcji dla zleceń oraz tworzenie agentów jednostek wykonawczych.

Celem agenta dyspozytora jest przesyłanie zleceń do jednostek wykonawczych oraz wybranie na podstawie ich odpowiedzi jednostki odpowiedzialnej za wykonanie zadania. Wiedzą agenta jest zestaw zleceń pozostałych do realizacji.

Jest on zdolny do wykonania szeregu akcji. Podstawową jest rozpoczęcie licytacji nowego zlecenia. Wszystkie istniejące w systemie jednostki wykonawcze są informowane o nadaniu nowego zlecenia. Następnie dystrybutor zbiera ich oferty dotyczące realizacji zlecenia. Po otrzymaniu wszystkich lub po określonym czasie kończy aukcję. Zakończenie aukcji jest kolejną akcją dystrybutora. Wszystkie oferty, które dotrą do niego po jej zakończeniu zostaną zignorowane. Ostatnią akcją dystrybutora jest wybór jednostki, która będzie odpowiedzialna za realizację właśnie wylicytowanego zlecenia. Otrzyma ona potwierdzenie przydziału. Pozostałe jednostki otrzymają odmowę.

**b) Jednostka Wykonawcza (*EUnit Agent*).**

Agent będący głową zespołu transportowego, zarządza trasą przejazdu i oblicza jego koszt.

Pierwszym celem agenta jest zebranie zespołu, wysyła on ofertę do elementów transportowych i z nich odpowiedzi wybiera najlepsze elementy. Kolejnym celem jest zrealizowanie zlecenia przy użyciu zdobytych zasobów.

Agent ten posiada wiedzę na temat już przydzielonych mu zleceń oraz składu posiadanej zespołu transportowego. Kolejnymi informacjami posiadanymi przez tego agenta są aktualnie licytowane zlecenie oraz położenie holonu.

Rozpoczęcie aukcji elementów holonu – Po utworzeniu i otrzymaniu zlecenia jednostka wykonawcza stara się zebrać zespół transportowy zdolny do jego realizacji. W tym celu powiadamia wszystkich zainteresowanych (elementy holonów niezwiązane z żadną jednostką). Po rozesłaniu zlecenia jednostka oczekuje na odpowiedzi. Z otrzymanych odpowiedzi jednostka wybiera te, które

najlepiej będą w stanie zrealizować dane zadanie. Następnie przesyła swoją ofertę do dystrybutora. Kolejną akcją jest odebranie odpowiedzi od dystrybutora i przekazanie tej informacji dalej. Jeśli odpowiedź jest twierdząca, jednostka powiadamia elementy o wyborze do holonu, w przeciwnym wypadku wszyscy otrzymują odpowiedź negatywną.

Po otrzymaniu odpowiedzi pozytywnej, jednostka może rozpoczęć realizację zlecenia, jednostka rozpoczyna realizację zlecenia od zebrania części holonu w jednym punkcie, zwykle jest to baza. Następnie udaje się do miejsca z którego ma zabrać towar. Kolejnym krokiem jest przewiezienie towaru do miejsca rozładunku. Zakończenie realizacji zlecenia odbywa się przez dotarcie do miejsca rozładunku i oczekanie na jego zakończenie. W tym momencie jednostka może rozpoczęć realizację kolejnego zlecenia z kalendarza lub wrócić do bazy. Oczywiście jednostka może realizować więcej niż jedno zlecenie jednocześnie jeśli pozwala jej na to pojemność naczepy.

**c) Kierowca (*Driver Agent*).**

Agent reprezentujący w systemie kierowcę, posiada informacje o jego położeniu i dotychczasowym czasie pracy. To on informuje system o zaistnieniu jego niedyspozycji (w systemie istnieją agenci pomocniczy, którzy wiedzą o tym fakcie wcześniej, jednak zaistnienie sytuacji kryzysowej jest oznajmiane przez poszczególnych agentów reprezentujących zasoby).

Cele agenta obejmują dołączenie do holonu i realizację zadania w czasie pobytu w nim. Na wiedzę posiadaną przez agenta składa się jego położenie, czas pracy oraz przynależność do holonu. Agent kierowcy ma możliwość odpowiadania na oferty dołączenia do holonu, przesyła wówczas swoją ofertę jednostce wykonawczej. Nie ma dla niego znaczenia czy jest to nowy holon czy też zamiana elementu. Kolejną akcją agenta jest wygenerowanie informacji o powstałej sytuacji kryzysowej i poinformowanie o tym jednostki wykonawczej.

**d) Ciężarówka (*Truck Agent*).**

Agent reprezentujący w systemie ciężarówkę. Jego rola sprowadza się do określania pozycji zasobu oraz informowania o sytuacjach kryzysowych. Celem tego agenta jest również dołączenie do holonu i realizacja wraz z nim zlecenia. Na

wiedzę agenta składa się jego położenie, moc (określająca jak duże przyczepy może ciągnąć) oraz fakt przynależności do holonu. Samodzielne akcje agenta są identyczne jak w przypadku kierowcy, czyli odpowiedź na ofertę przyłączenia do holonu oraz poinformowanie o sytuacji kryzysowej.

#### e) Naczepa (*Trailer Agent*).

Jest reprezentacją naczepy, określa jej ładowność. Do jej zadań należy określanie pozycji, dotychczasowego załadowania i informowanie o zaistnieniu sytuacji kryzysowych. Cele agenta pokrywają się z celami dwóch pozostałych elementów transportowych, czyli jest to dołączenie do zespołu, realizacja zlecenia i informowanie o sytuacjach kryzysowych. Na wiedzę składa się ładowność, położenie i dotychczasowe zapełnienie, a także fakt przynależności do holona. W tym przypadku dostępne dla agenta akcje również są takie same jak w przypadku pozostałych.

#### f) Inne

Ponadto w systemie występują agenci pomocniczy niezwiązani bezpośrednio ze znajdowaniem rozwiązania. **Info Agent** jest agentem informacyjnym, zajmuje się tworzeniem innych agentów oraz przechowywaniem informacji na ich temat. **GUIAgent** jest agentem odpowiedzialnym za wygenerowanie interfejsu użytkownika oraz za przekazywanie informacji pomiędzy tymże interfejsem a resztą systemu. Posiada on wiedzę na temat zleceń oraz czasu ich napłygnięcia do realizacji.

**Tabela – Cechy, cykl życia i cele agentów**

Agent	Cykl życia	Cele
Ciągnik ( <i>Truck</i> )	Agenci ciągnika pojawiają się w systemie po jego załadowaniu(plik konfiguracyjny) i trwają, aż do jego zakończenia.	Dołączenie do zespołu transportowego, realizacja zlecenia
Naczepa ( <i>Trailer</i> )	Agenci naczepy pojawiają się w systemie po jego załadowaniu(plik konfiguracyjny) i trwają, aż do jego zakończenia.	Dołączenie do zespołu transportowego, realizacja zlecenia

Kierowca ( <i>Driver</i> )	Kierowcy pojawiają się w systemie po jego załadowaniu(plik konfiguracyjny) i trwają, aż do jego zakończenia..	Działanie do zespołu transportowego, realizacja zlecenia
Dystrybutor ( <i>Distributor</i> )	Dystrybutor zleceń jest w systemie cały czas od momentu jego uruchomienia.	Przyjmowanie zleceń Wyznaczenie jednostki wykonawczej do realizacji zlecenia
Jednostka wykonawcza ( <i>eUnit</i> )	Jednostka wykonawcza pojawia się w systemie, gdy dystrybutor otrzymuje nowe zadanie, zakończona została procedura jego negocjacji. Zostaje ona wyznaczona do tego zadania i pozostaje w systemie aż do jego realizacji.	Znalezienie optymalnego sposobu realizacji zlecenia Rozwiązywanie sytuacji kryzysowych

### 2.5.2. Współpraca agentów

Współpraca pomiędzy agentami polega na zestawianiu zespołów transportowych potrzebnych do wykonania zleceń i wspólnej ich realizacji, a także minimalizacji kosztów wykonania danego zestawu zleceń.

**Holony**, to struktury złożone z agentów, postrzegane przez resztę systemu jako jeden agent. W przypadku naszego systemu holonem będzie zespół transportowy.

**Model holoniczny.** Zastosowany przez nas model zakłada, że agenci porzucają jedynie część swojej niezależności. Cały holon jest reprezentowany przez głowę, wybranego agenta, który komunikuje się z całą społecznością. Jego kompetencje mogą być różne, od zadań czysto administracyjnych do wydawania bezpośrednich poleceń pozostałym elementom holonu. Ponadto głowa ma prawo przydzielania zasobów, planowania, a także negocjowania z innymi agentami na podstawie planów i celów agentów składowych. Jednocześnie zadaniem głowy jest wymiana elementów holonu lub też ich odłączanie i przyłączanie. W naszym przypadku głową holonu stanowi Jednostka Wykonawcza (*EUnit Agent*).

W zastosowanym przez modelu uwzględniono koncepcję zobowiązań. Podejście takie pozwoliło zagwarantować dostęp do zasobów. Agenci przyłączając się do holonu, podejmują zobowiązanie do wykonania zadania. Nie mogą wówczas podjąć kolejnego aż do czasu, gdy

zostaną zwolnieni przez głowę holonu. Realizacja tego podejścia jest dokładniej opisana w punkcie ‘Budowanie’.

**Budowanie.** W przypadku naszego systemu, holony są tworzone w momencie zakończenia negocjacji i uzyskania wyniku dla danego zlecenia. Na początek tworzona jest jednostka wykonawcza, która ma zostać jego głową. Nie posiada ona żadnych elementów holonu, przez co musi on być stworzony od podstaw. Wyborem elementów zajmuje się procedura negocjacyjna. Sam EUnit nie bierze w tym procesie udziału.

Jeśli holon zostanie utworzony, jego elementy stracą część swojej autonomii. Nie będą mogły odbierać informacji o nowych zleceniach od jednostek wykonawczych innych niż jednostka będąca głową ich holonu, nie będą mogły również wysyłać odpowiedzi na takie zlecenia. Pozostanie im możliwość generowania sytuacji kryzysowych.

**Nowe zlecenie.** Gdy jednostka wykonawcza będąca głową holonu bezczynnego (w jej kalendarzu nie znajduje się żadne zadanie) otrzymuje zlecenie, sprawdzana jest konfiguracja. Jeśli holon jest w stanie obsłużyć dane zlecenie, podejmuje się tego zadania. W przeciwnym razie je odrzuca.

### 3. Działanie systemu

#### 3.1. Przypadki użycia

Opisane zostały tutaj przypadki użycia wraz z ich warunkami wstępymi, wykonywanymi czynnościami oraz ich rezultatem.

a) **Wczytanie konfiguracji.** Przypadek opisujący wczytanie konfiguracji z pliku.

Celem tej operacji jest dostarczenie do systemu zestawu zleceń, który ma zostać obsłużony w czasie tej symulacji oraz innych niezbędnych do działania danych.

Opis
Aktor wczytuje konfiguracje z pliku
Wymagania
-
Wykonywane czynności

1. Wybór pliku
2. System wczytuje plik i ładuje zawartą w nim konfigurację
<b>Wynik z punktu widzenia aktora</b>
Konfiguracja znajduje się w systemie
<b>Efekt w przypadku niepowodzenia</b>
Konfiguracja nie znajduje się w systemie, aktor zostaje powiadomiony o przyczynie błędu odpowiednim komunikatem

**b) Przeliczenie zestawu zleceń na dynamiczny.** Wczytywane zestawy zleceń są statycznymi testami, to oznacza, że wszystkie zlecenia napływają do dystrybutora w tym samym momencie, na początku symulacji. Przeliczenie na zestaw dynamiczny pozwala na zmianę czasu przyjścia zleceń do dystrybutora przy zachowaniu możliwości jego wykonania w wyznaczonym czasie.

<b>Opis</b>
Aktor wybiera rodzaj przeliczenia na dynamiczny zestaw zleceń
<b>Wymagania</b>
Przygotowanie grafu sieci komunikacyjnej
<b>Wykonywane czynności</b>
<ol style="list-style-type: none"> <li>1. Wybór rodzaju dynamiki w zestawie</li> <li>2. Zatwierdzenie ustawień</li> <li>3. System przelicza wczytany zestaw na dynamiczny</li> <li>4. Wyświetlone zostają wyniki</li> </ol>
<b>Wynik z punktu widzenia aktora</b>
Czasy nadejścia zleceń zostają rozproszone w czasie

**c) Przeprowadzenie obliczeń.** Zasadniczym etapem działania systemu jest znalezienie rozwiązania postawionego problemu. W tym celu należy uruchomić

symulację i przydzielić w ten sposób zadania poszczególnym zespołom transportowym, które zostaną utworzone w czasie jej trwania.

<b>Opis</b>
Aktor uruchamia symulację dla statycznego zestawu zleceń
<b>Wymagania</b>
Zdefiniowanie zasobów
<b>Wykonywane czynności</b>
<ol style="list-style-type: none"> <li>1. Rozpoczęcie symulacji</li> <li>2. Uruchomienie zegara symulacji</li> <li>3. System znajduje rozwiązania</li> </ol>
<b>Wynik z punktu widzenia aktora</b>
Znalezione zostało rozwiązanie postawionego problemu

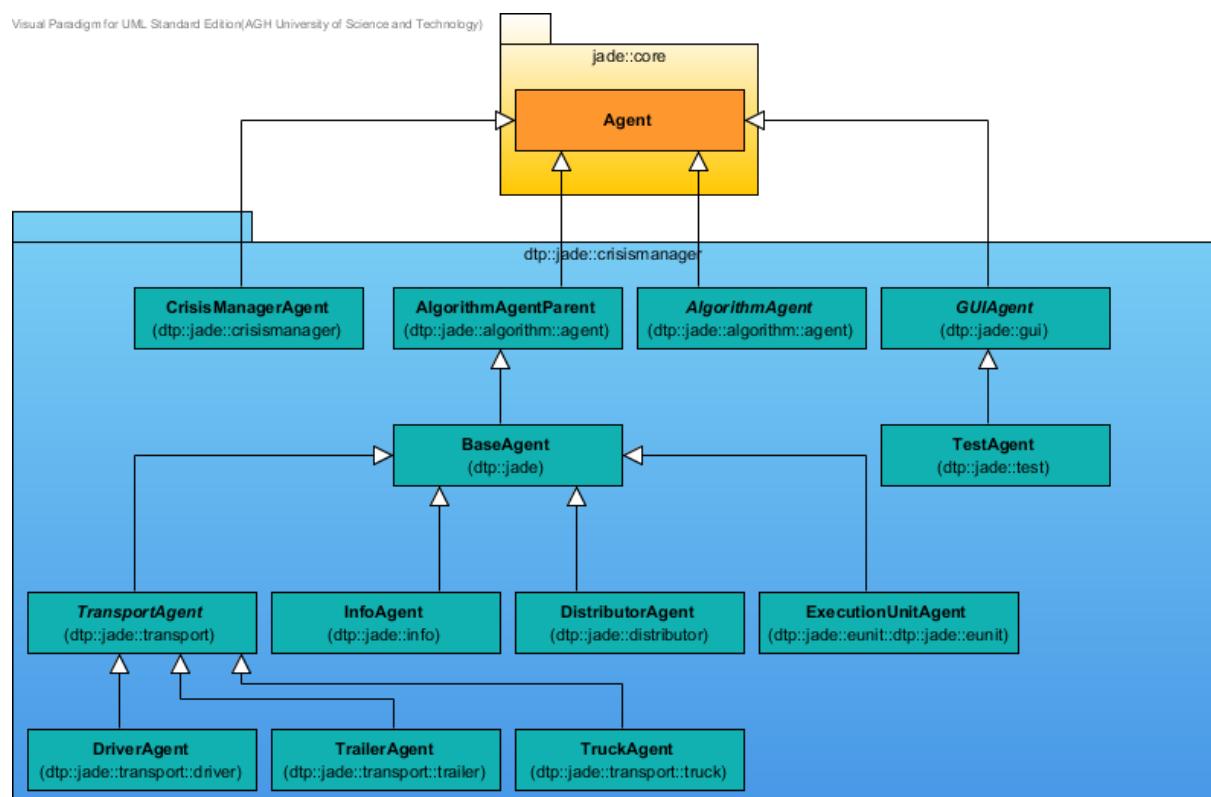
d) **Przeprowadzenie obliczeń dynamicznych.** Uruchomienie symulacji dynamicznej z punktu widzenia użytkownika różni się tylko wymogiem przeliczenia zestawu na dynamiczny. Jednak jej przebieg jest podzielony na kolejne kroki czasowe.

<b>Opis</b>
Aktor uruchamia dynamiczną symulację
<b>Wymagania</b>
<ol style="list-style-type: none"> <li>1. Zdefiniowanie zasobów</li> <li>2. Przeliczenie zestawu zleceń na dynamiczny</li> </ol>
<b>Wykonywane czynności</b>
<ol style="list-style-type: none"> <li>1. Ustalenie prędkości symulacji</li> <li>2. Rozpoczęcie symulacji</li> <li>3. Uruchomienie zegara symulacji</li> <li>4. System przechodzi kolejne kroki symulacji przekazując odpowiednie zlecenia do dystrybutora</li> </ol>

5. System znajduje rozwiązanie
<b>Wynik z punktu widzenia aktora</b>
Znalezione zostało rozwiązanie postawionego problemu

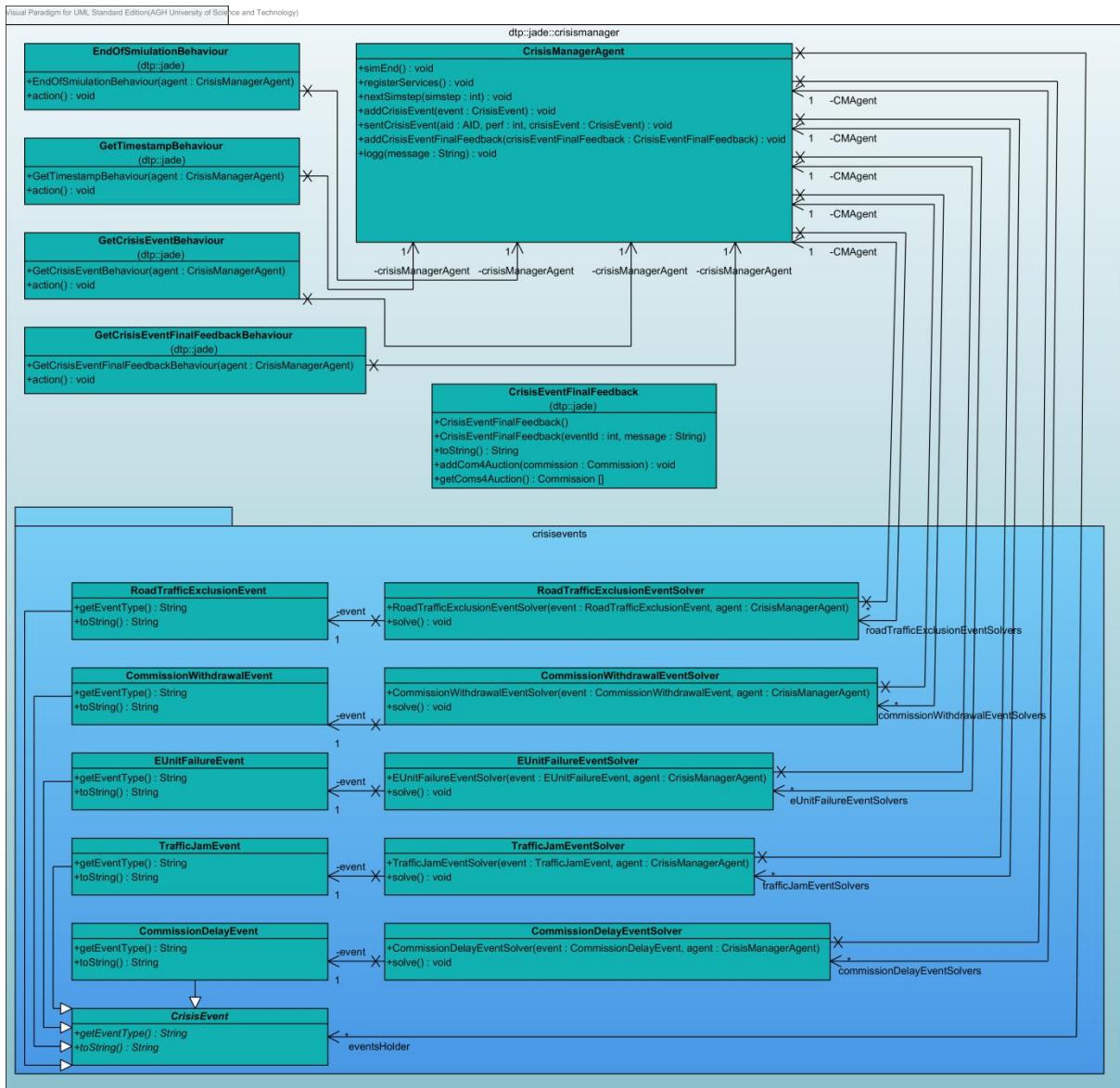
### 3.2. Agenci w systemie – diagramy

W rozdziale tym przedstawiono diagramy obrazujące zaimplementowane pakiety z agentami działającymi w systemie.

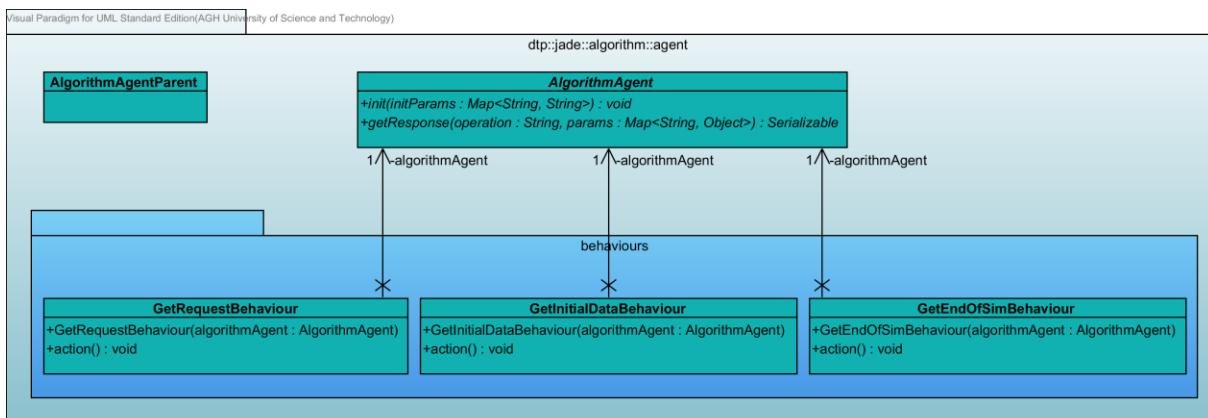


Rysunek 2 Diagram klas przedstawiający relacje agentów w systemie

## Dispatch Rider – dokumentacja projektowa

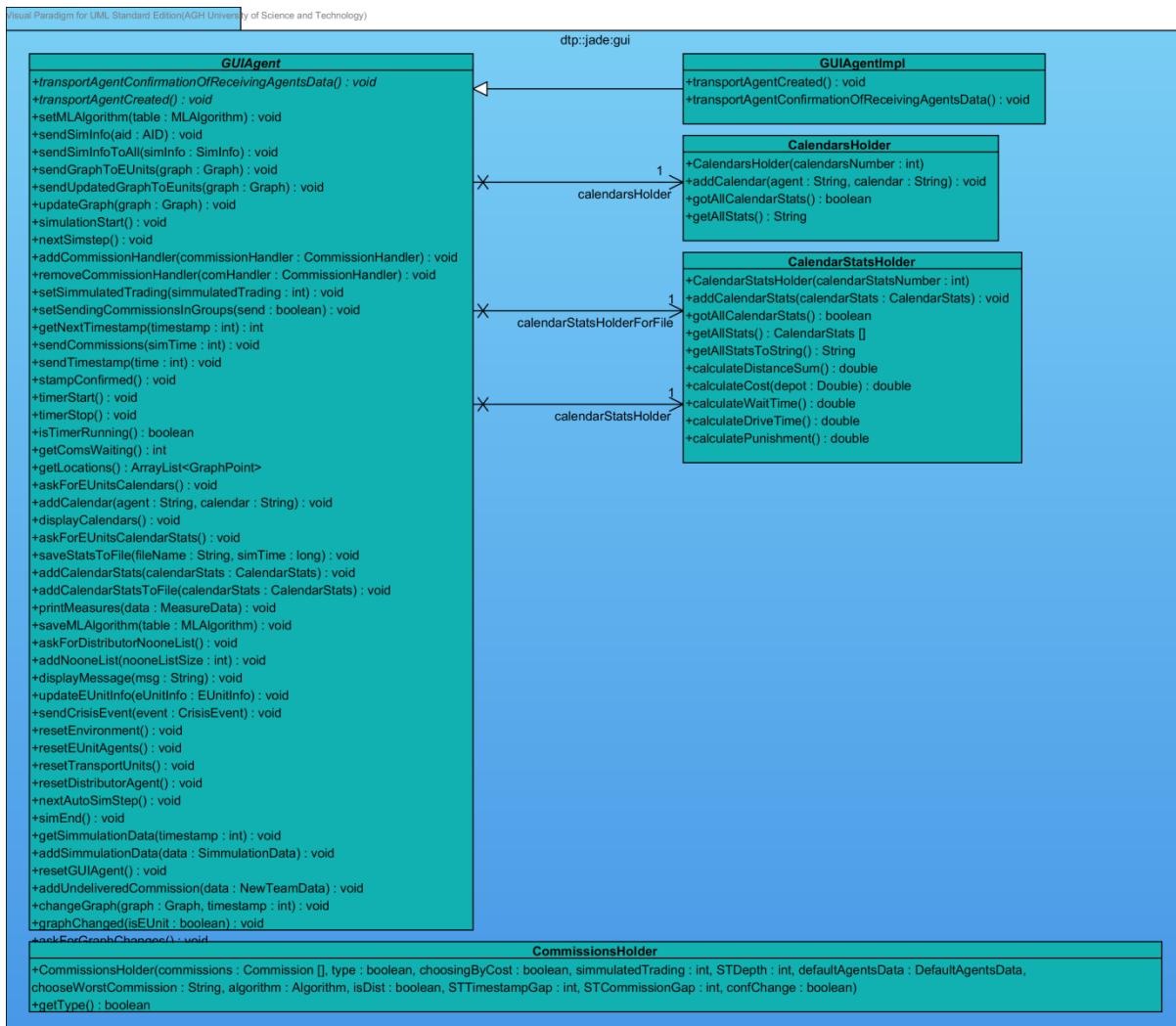


Rysunek 3 Diagram klas przedstawiający mechanizm agenta zarządzającego sytuacjami kryzysowymi



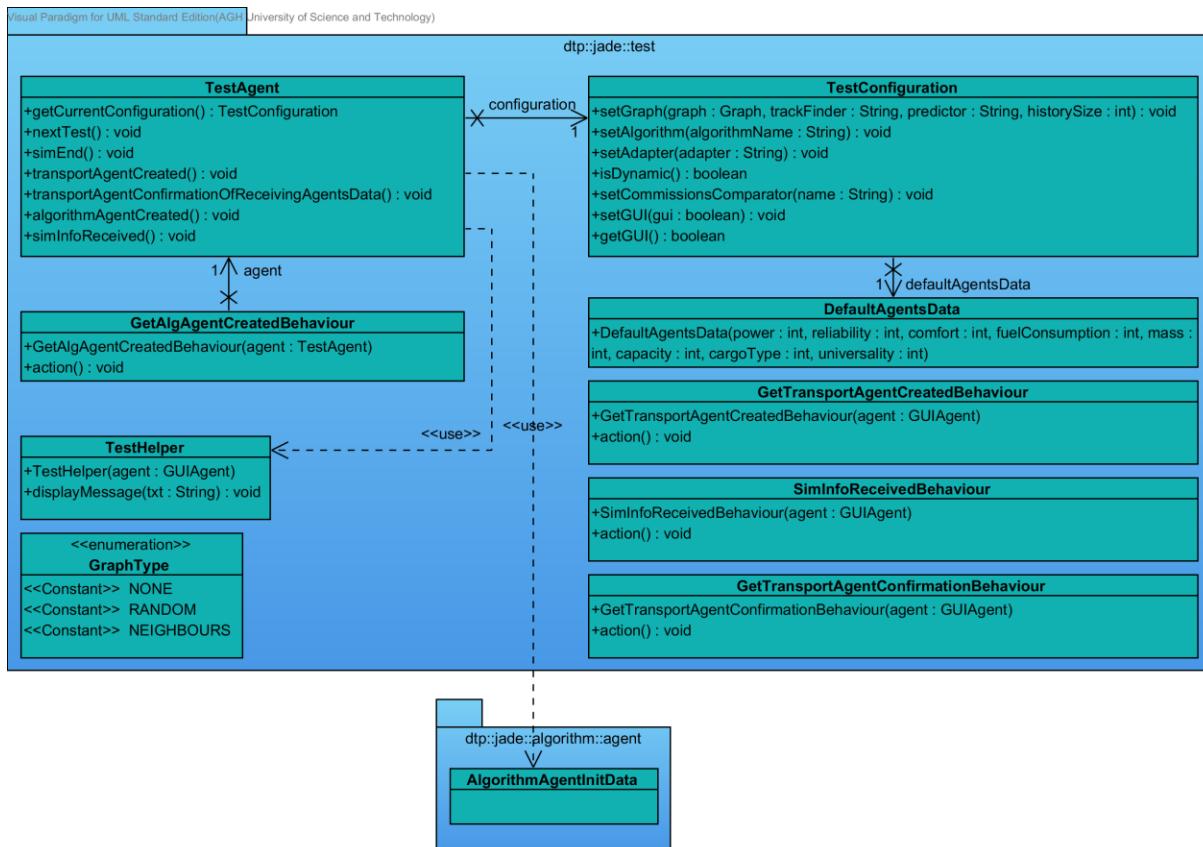
Rysunek 4 Diagram klas przedstawiający mechanizm AlgorithmAgenta

## Dispatch Rider – dokumentacja projektowa

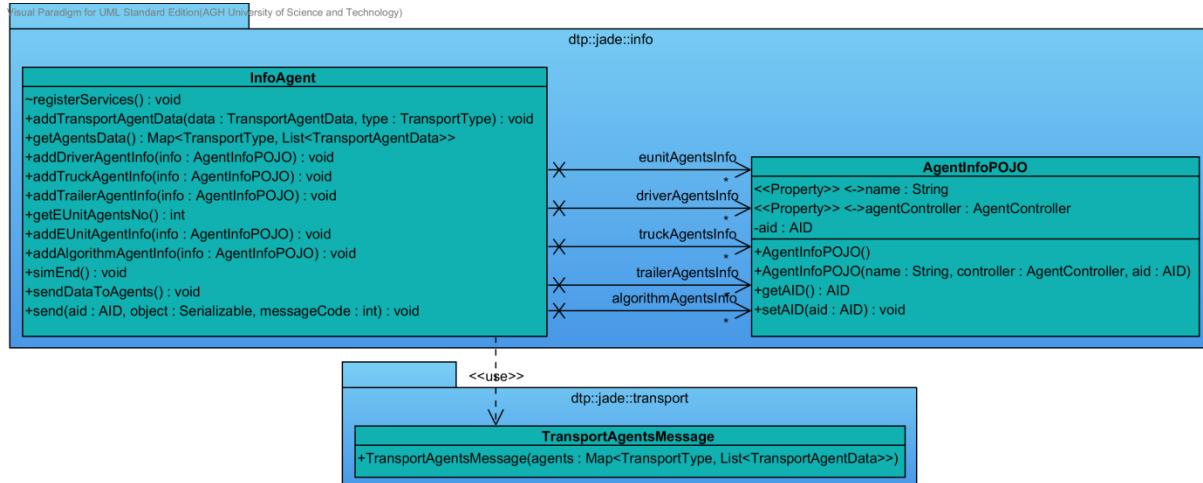


Rysunek 5 Diagram klas przedstawiający mechanizm GUIAgenta

## Dispatch Rider – dokumentacja projektowa



Rysunek 6 Diagram klas przedstawiający mechanizm TestAgenta



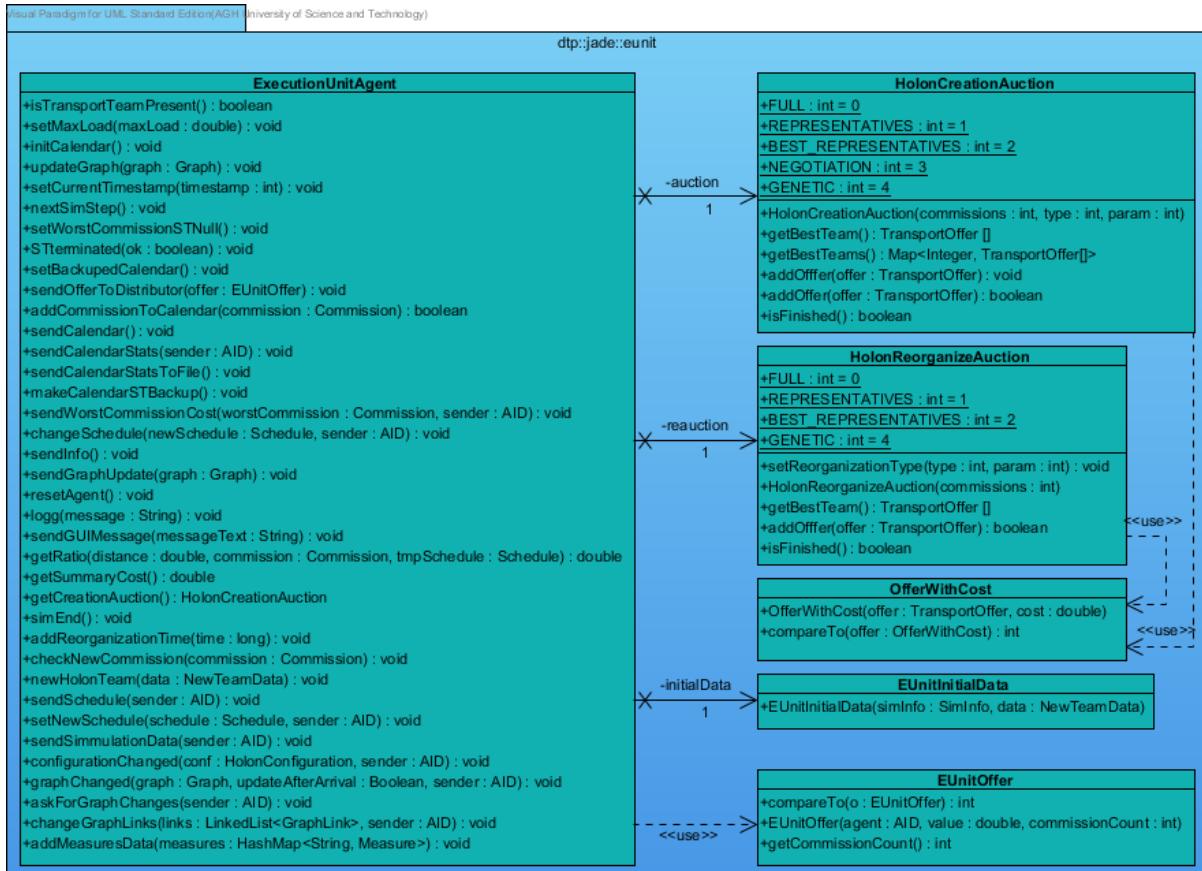
Rysunek 7 Diagram klas przedstawiający mechanizm InfoAgenta

## Dispatch Rider – dokumentacja projektowa

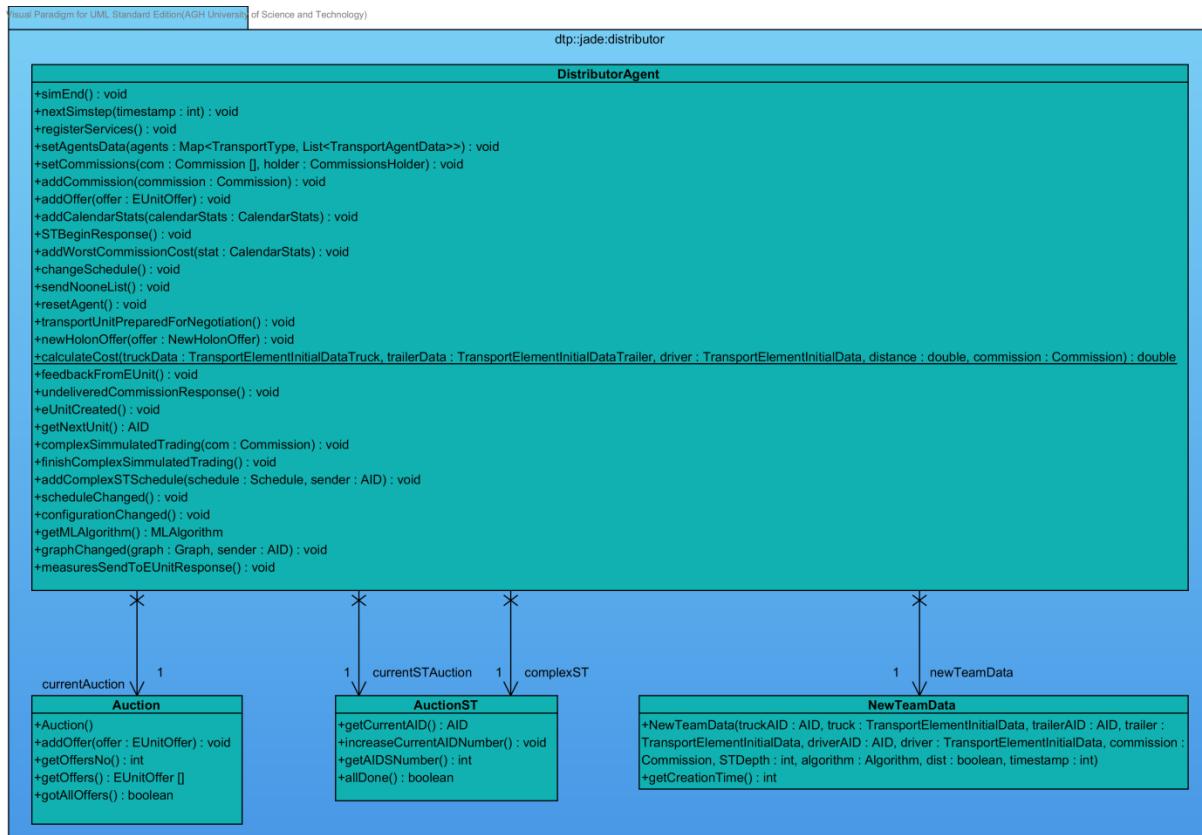


Rysunek 8 Diagram klas przedstawiający mechanizm BaseAgenta

## Dispatch Rider – dokumentacja projektowa

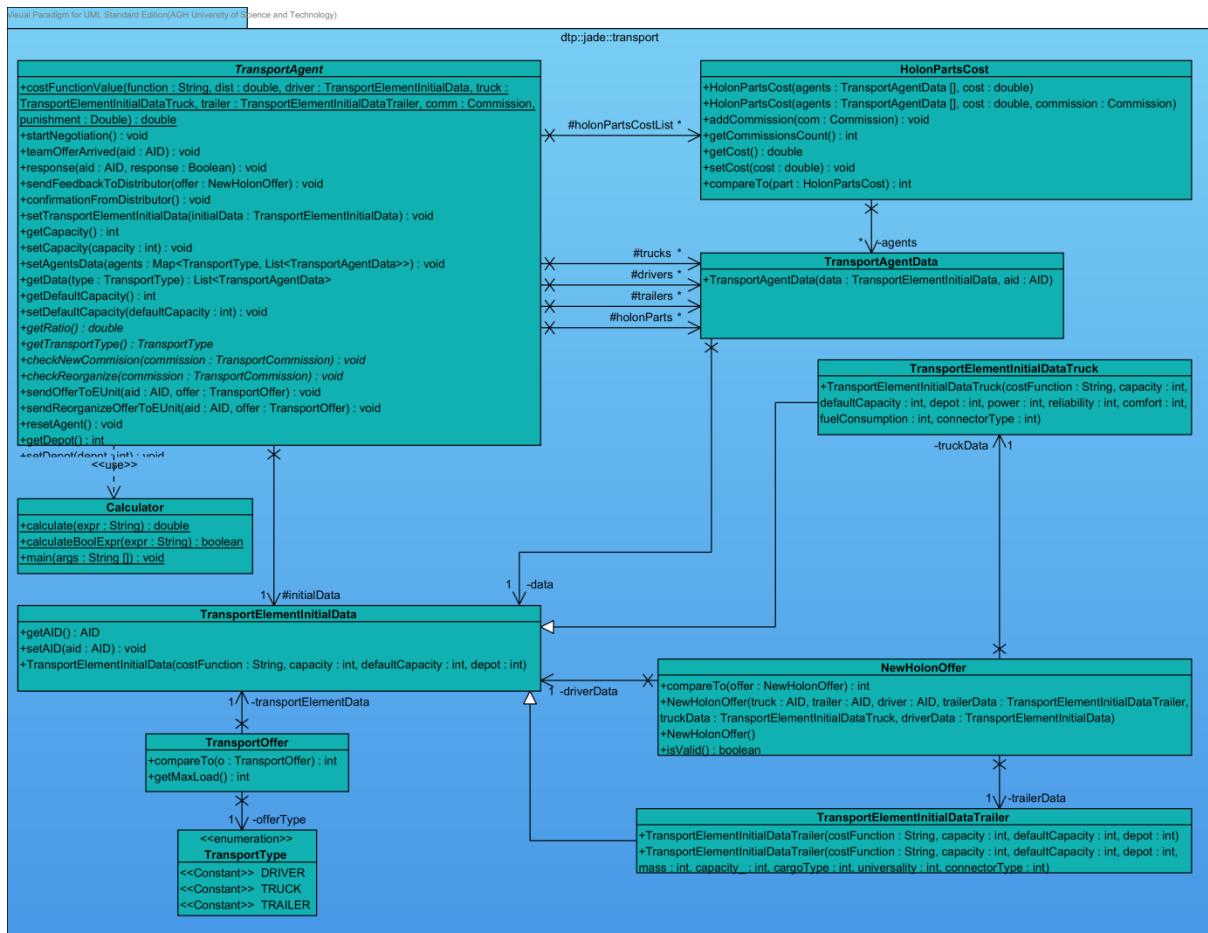


Rysunek 9 Diagram klas przedstawiający mechanizm ExecutionUnitAgenta

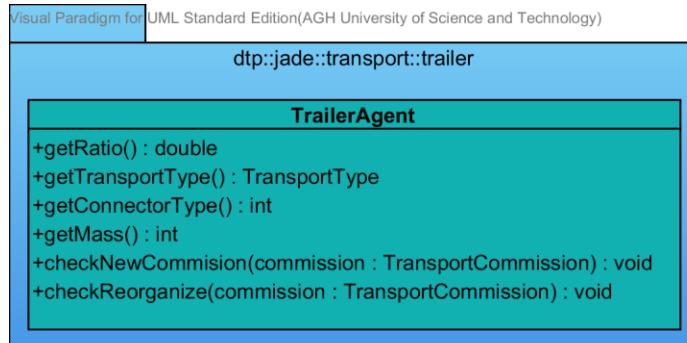


Rysunek 10 Diagram klas przedstawiający mechanizm DistributorAgenta

## Dispatch Rider – dokumentacja projektowa

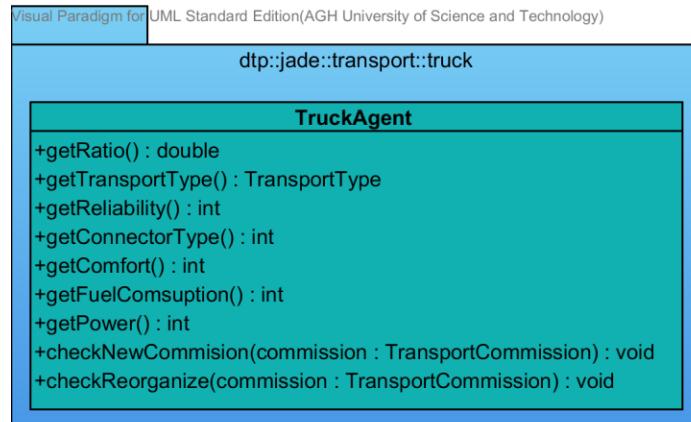


Rysunek 12 Diagram klas przedstawiający mechanizm TransportAgenta

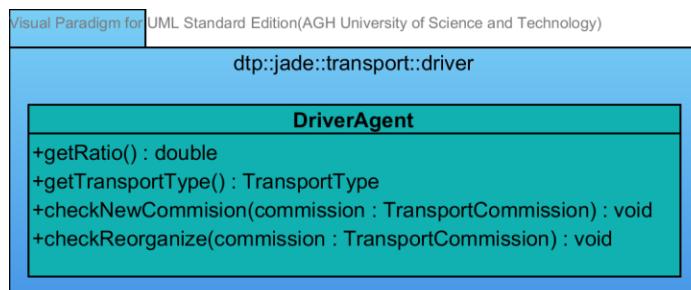


Rysunek 11 Diagram klas przedstawiający mechanizm TrailerAgenta

## Dispatch Rider – dokumentacja projektowa

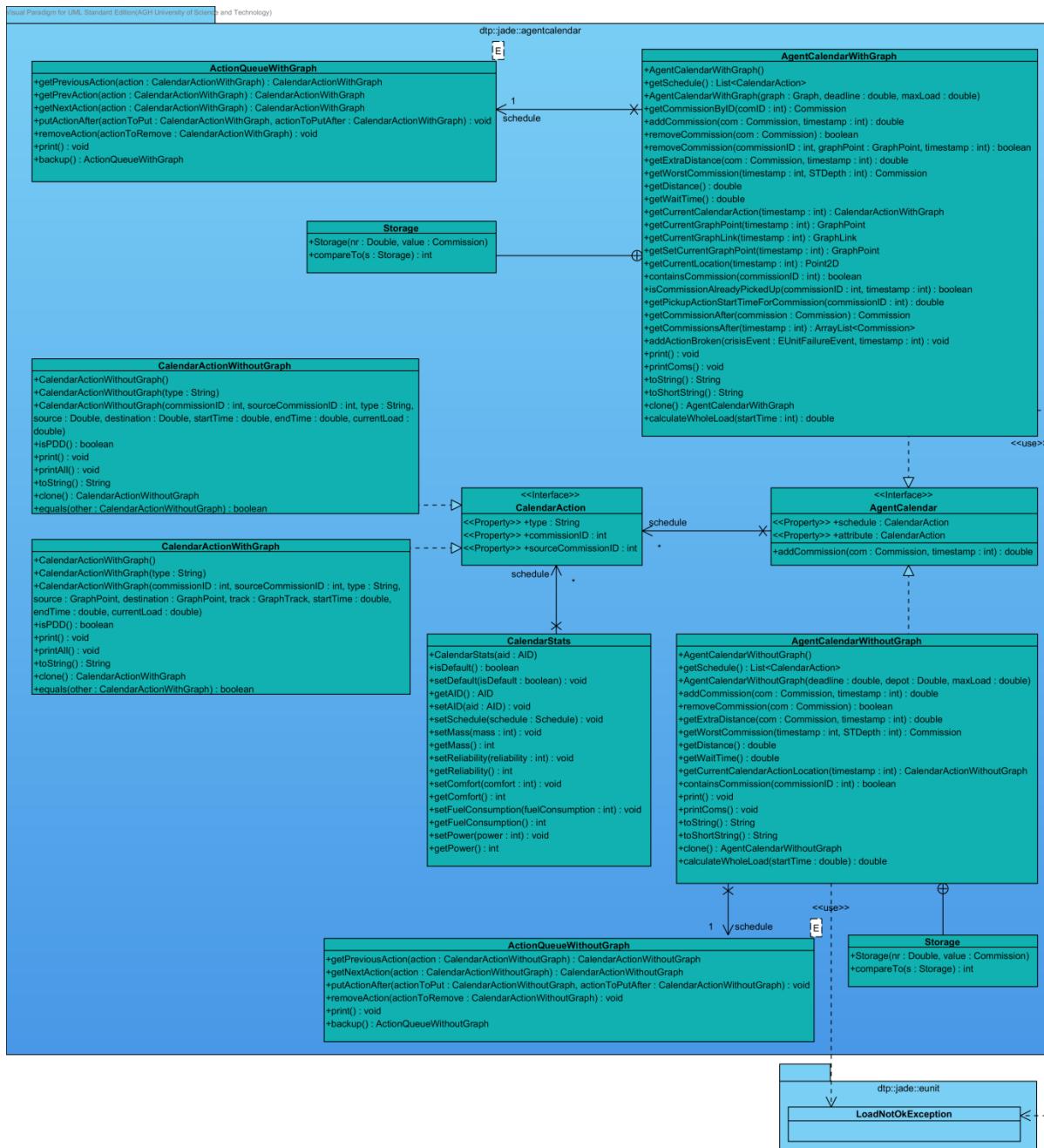


Rysunek 13 Diagram klas przedstawiający mechanizm  
TruckAgenta



Rysunek 14 Diagram klas przedstawiający mechanizm  
DriverAgenta

# Dispatch Rider – dokumentacja projektowa



Rysunek 15 Diagram przedstawiający mechanizm zarządzania harmonogramem

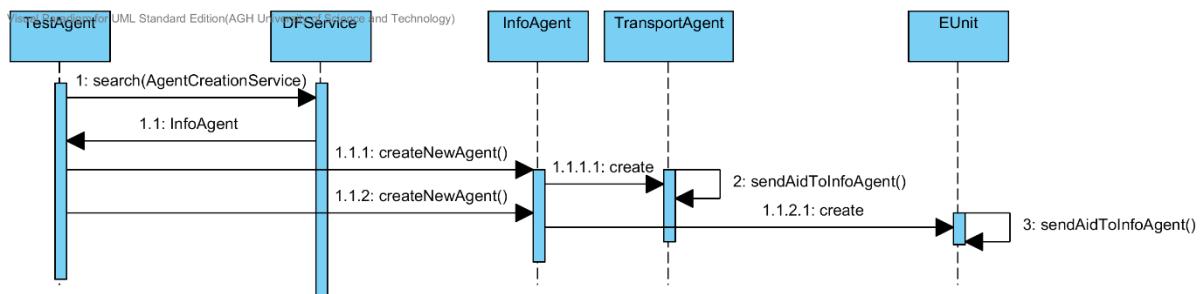
### 3.3. Poglądowy opis symulacji

Wyróżniamy następujące etapy symulacji (w kolejności chronologicznej):

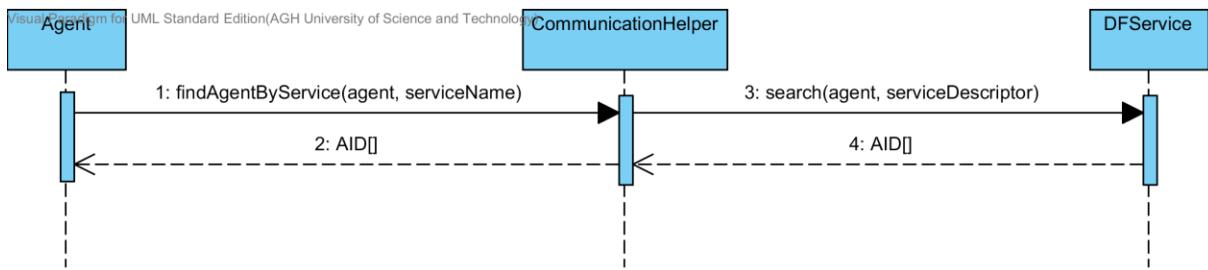
- Tworzeni są agenci główni: `TestAgent`, `DistributorAgent`, `CrisisManager`, `Info-Agent` oraz agenci pomocniczy
- Wczytywane są odpowiednie parametry symulacji z plików konfiguracyjnych
- Tworzeni są agenci składowi (Kierowcy, Ciągniki i Przyczepy)
- Agenci składowi i Dystrybutor otrzymują informacje o wszystkich agentach (składowych) w systemie. Informacje te są zapisywane do późniejszego użycia

## Dispatch Rider – dokumentacja projektowa

- Rozpoczyna się wysyłanie zleceń i procedura ich obsługi. Wysyłanie zleceń inicjuje proces negocjacji. W trakcie obsługi zleceń tworzone są EUnity (jak również wypożyczane jednostki, w przypadku, gdy jakieś zlecenie nie może zostać zrealizowane)
- Wyniki są zapisywane do odpowiednich plików



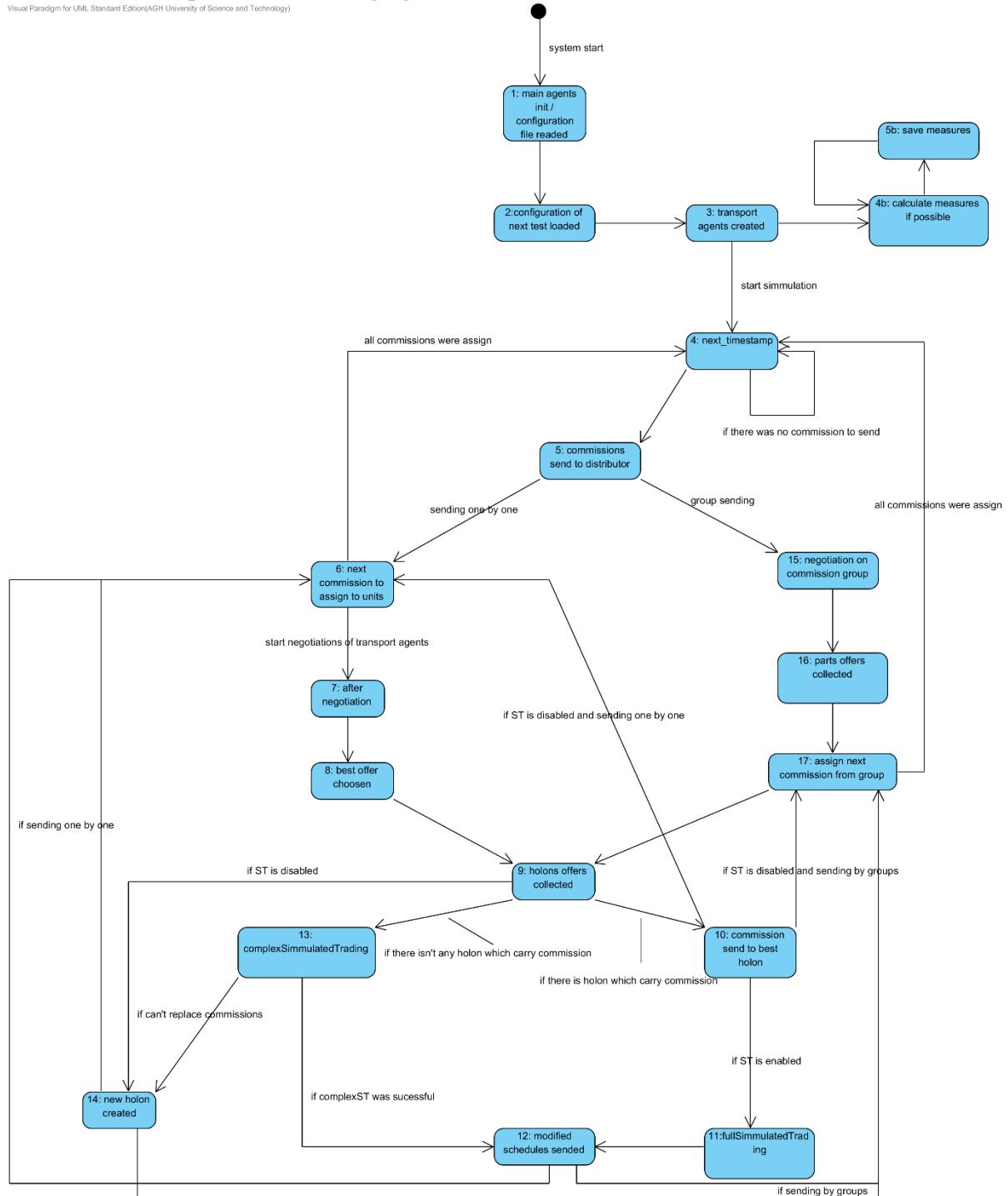
Rysunek 16 Diagram sekwencji przedstawiający tworzenie agentów



Rysunek 17 Diagram sekwencji przedstawiający wyszukiwanie serwisów

### 3.3.1. Krótki opis architektury systemu

Visual Paradigm for UML Standard Edition/AGH University of Science and Technology



Rysunek 18 Diagram maszyny stanowej obrazujący działanie systemu

System składa się z następujących elementów:

- Sterowanie symulacją (TestAgent) – komponent odpowiedzialny za wczytywanie plików wejściowych, oraz generowanie wyjściowych. Jego zadaniem jest rozpoczęcie (inicjalizacja) symulacji. W związku z tym komunikuje się on z InfoAgentem i zleca mu stworzenie agentów części, którzy potem będą tworzyć holony. Ponadto jak sama nazwa

wskazuje, element ten odpowiada za sterowanie, tzn. to on decyduje o postępie symulacji (wysyła kolejne znaczniki czasowe). Dodatkowo pełni funkcję „świata zewnętrznego” z punktu widzenia innych agentów. Odpowiada za przesyłanie napływających zleceń, jak również dba o propagację zmian w grafie sieci drogowej.

Oprócz tego ma również możliwość stworzenia początkowej konfiguracji wykorzystywanych algorytmów samodzielnie, na podstawie analizy problemu, który ma być rozwiązany (pliku z opisem benchmarku)

- Dane o agentach w systemie (InfoAgent) – agent, który zajmuje się tworzeniem (z wykorzystaniem mechanizmów platformy JADE) agentów części, jak również jednostek wykonawczych. Oprócz tego przechowuje identyfikatory tworzonych agentów
- Agenci części (Kierowcy, Naczepy i Ciągniki) – są elementami tworzącymi jednostkę wykonawczą. O ich wzajemnym połączeniu decydują sami, w procesie negocjacji, gdzie każdy agent bierze pod uwagę własne interesy, reprezentowane w postaci definiowanych przez użytkownika funkcji (każdy agent może posiadać różne funkcje, nawet w ramach tej samej grupy).
- Jednostki wykonawcze (Eunit) – zajmuje się wykonywaniem zleceń, przydzielanym przez Dystrybutora. Po otrzymaniu nowego zlecenia oblicza koszt jego realizacji i odsyła Dystrybutorowi. Dodatkowo przekazuje informacje o swoim stanie do modułu sterowania symulacją, oraz do Dystrybutora. Oprócz tego ma również możliwość zmiany swojej konfiguracji (sposobu działania) w trakcie trwania symulacji.
- Dystrybucja zleceń (Dystrybutor) – otrzymuje napływające zlecenia od jednostki sterującej. Zlecenia są następnie ustawiane w optymalnej kolejności (różne polityki), w taki sposób, żeby zwiększyć prawdopodobieństwo utworzenia optymalnych tras po przydzieleniu zleceń jednostkom wykonawczym. Każde zlecenie jest wysyłane do istniejących jednostek wykonawczych z zapytaniem o koszt ich realizacji. Wybierana jest jednostka oferująca najmniejszy koszt. Po przydzieleniu zlecenia Dystrybutor inicjuje procedurę Simulated Trading, pobierając od jednostek wykonawczych ich kalendarze. Po przeprowadzeniu Simulated trading odsyła zmodyfikowane kalendarze z powrotem. W przypadku, gdy zlecenie nie może zostać zrealizowane inicjuje proces negocjacji, wybiera najlepszą konfigurację agentów i zleca utworzenie nowej jednostki wykonawczej. Jeśli zlecenie nie może zostać zrealizowane przez żadną jednostkę, to uruchamiane jest *full*

*Simulated Trading.* Dodatkowo zajmuje się wyliczaniem miar, wraz z dynamiczną zmianą konfiguracji, bazując na statycznych regułach, bądź na uczeniu maszynowym (Q-Learning).

- Agenci pomocniczy – agenci pomocniczy służą do wykonywania skomplikowanych obliczeń.

Dodatkowo system mam możliwość wizualizacji:

- wizualizacja miar – wizualizacja miar może odbywać się w trakcie działania symulacji, lub po jej zakończeniu, z wykorzystaniem zapisanych wartości miar w jednym z plików wynikowych
- wizualizacja przebiegu symulacji – jest realizowana na podstawie pliku z zapisem symulacji

### 3.4. Tworzenie ofert do powstania nowego holonu

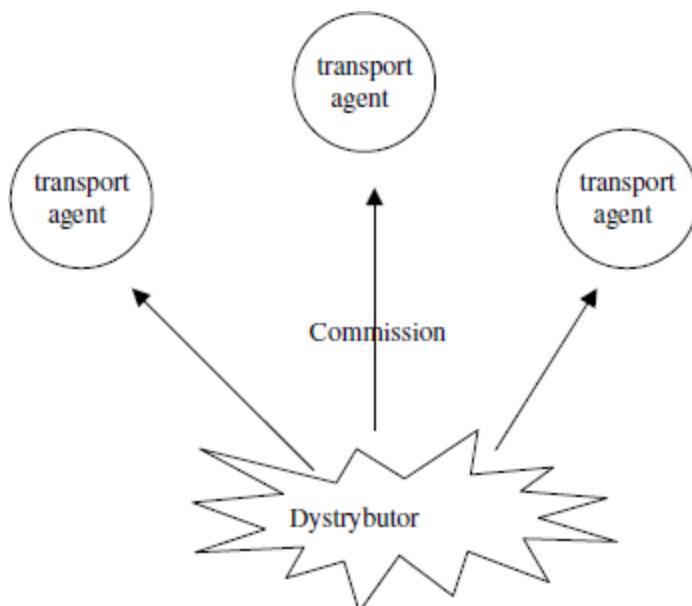
Koncepcja zakłada, że agenci sami łączą się ze sobą wykorzystując swoje indywidualne preferencje (funkcje kosztu). Kolejne podrozdziały opisują ten proces.

#### 3.4.1. Procedura negocjacji

Stanowi ona podstawę tworzenia holonów. Składa się z kilku części, z których większość dzieje się współbieżnie.

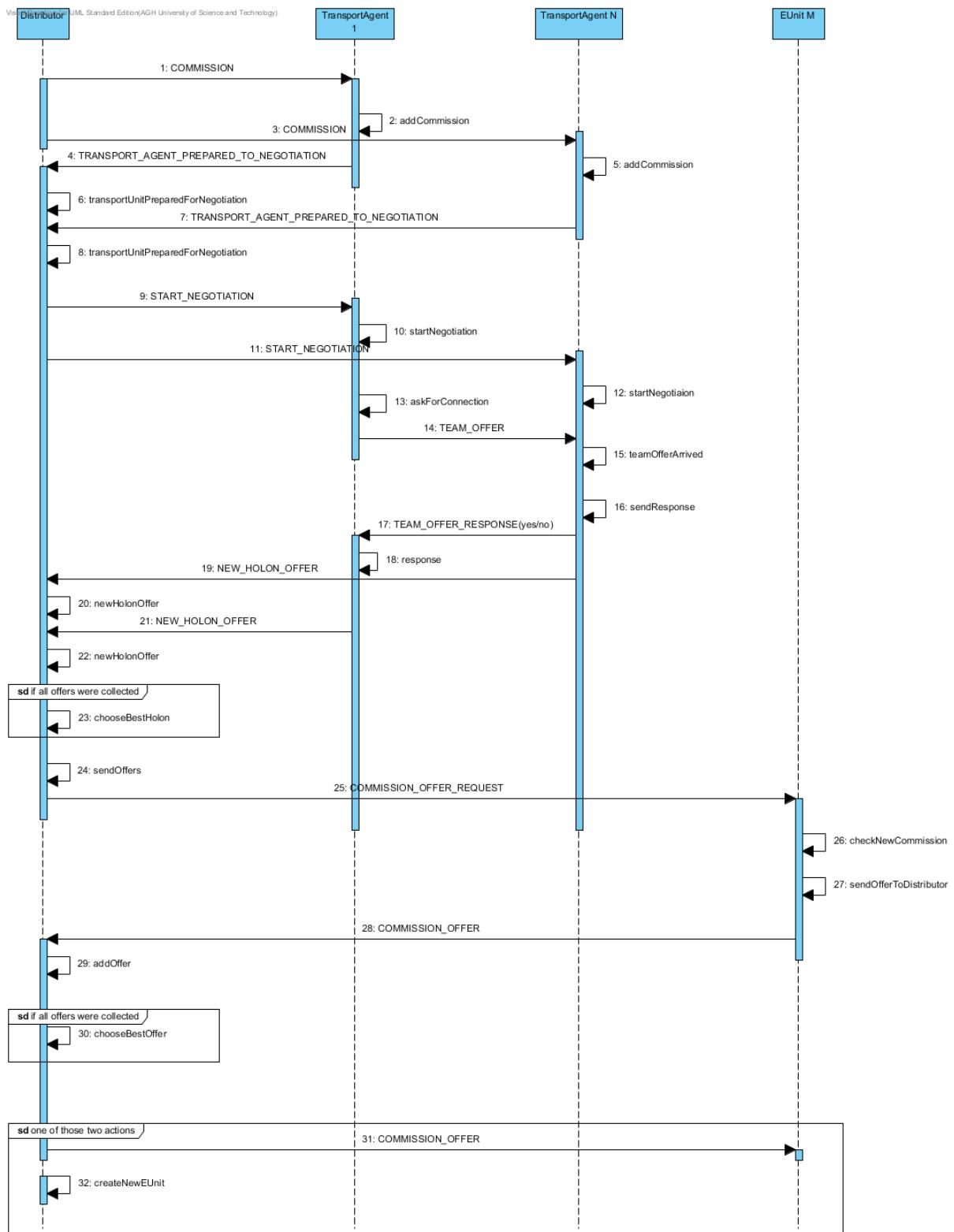
##### Przygotowanie środowiska do przeprowadzenia negocjacji:

- a) Dystrybutor rozsyła kolejne zlecenie do wszystkich agentów składowych (na tym etapie wszyscy agenci są równoprawnii)



Rysunek 19 Rozsyłanie zlecenia do agentów

## Dispatch Rider – dokumentacja projektowa



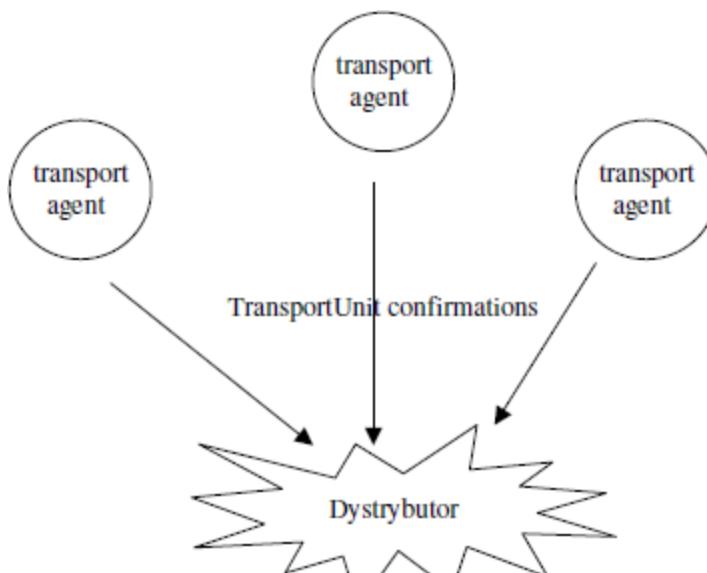
**Rysunek 20 Diagram sekwencji obrazujący obsługę nadchodzącego zlecenia**

- b) Agenci wykorzystując swoją funkcje kosztu, dane o innych agentach, oraz parametry zlecenia tworzą swoje listy preferencji(wartości funkcji kosztu po

podstawieniu wspomnianych parametrów są minimalizowane, tzn. na pierwszych miejscach będą pary, dla których wartość funkcji kosztu danego agenta jest najmniejsza). Listy te zawierają pary identyfikatorów AID, która są pewnego rodzaju adresem agentów w platformie JADE. Na przykład dla kierowcy przykładowa lista mogłaby wyglądać następująco:

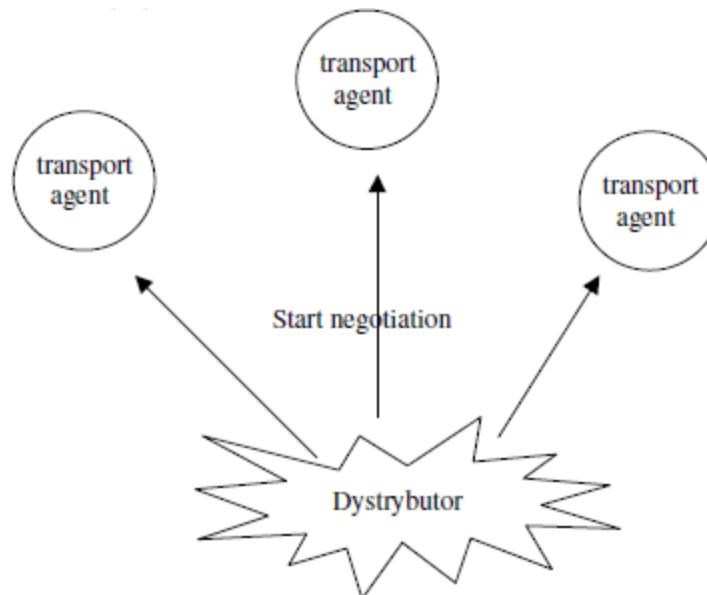
Truck	Trailer
Truck #1	Trailer #0
Truck #0	Trailer #15
...	...

- c) Kiedy wszyscy agenci utworzą swoje listy preferencji, powiadamiają o tym Dystrybutora.



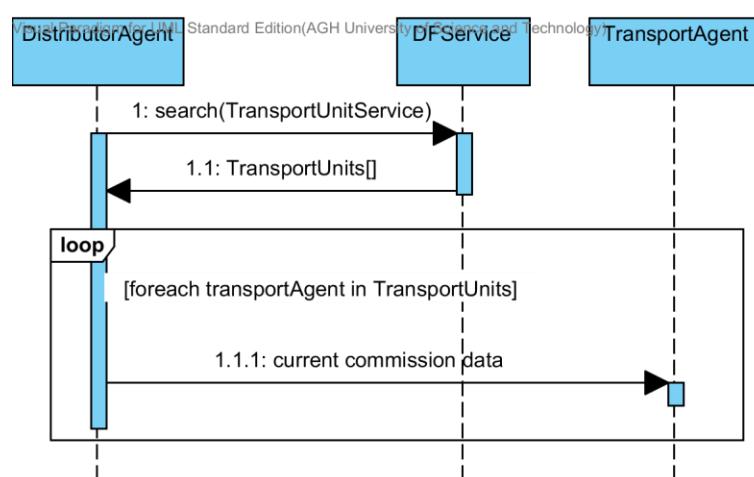
Rysunek 21 Wysywanie odpowiedzi TransportAgentów

- d) Dystrybutor po zebraniu wszystkich powiadomień wysyła agentom sygnał do rozpoczęcia negocjacji.



### Przebieg negocjacji:

Z uwagi na to, że procedura negocjacji jest dość złożona i w pełni współbieżna, ciężko byłoby ją opisać w odniesieniu do wszystkich agentów. Poniżej przedstawiam algorytm działania pojedynczego agenta w dużym uproszczeniu (z uwagi na silną współbieżność poniżej znajduje się opis poszczególnych faz negocjacji, jednak w rzeczywistości ich kolejność nie musi być zachowana). Oprócz tego dochodzą sprawy związane z buforowaniem zapytań i zapobieganiu zakleszczeniom), oraz przykład ilustrujący wycinek z przykładowej negocjacji.



Rysunek 23 Diagram sekwencji obrazujący procedurę negocjacji

- Rozpoczęcie negocjacji – na początku agent sprawdza, czy jest częścią holonu, jeżeli tak, to na wszystkie zapytania odpowiada „no”, oraz od razu wysyła pustą ofertę do Dystrybutora. Każdy agent, który nie jest częścią holonu, usiłuje połączyć się z innymi, w jak najkorzystniejszy sposób. Do tego celu wykorzystuje listę preferencji. Bierze pierwszy wpis i wysyła zapytanie do agentów, którzy się w nim znajdują.
- Wysyłanie zapytania o połączenie - zapytanie wysyłane jest przy użyciu AID zapisanego w liście
- Przyjmowanie zapytań - po przyjęciu zapytania agent sprawdza, czy ma pytającego na szczytce listy preferencji. Jeżeli tak to odpowiada mu „yes” i modyfikuje swoją listę preferencji pozostawiając tylko wpisy dotyczące

agenta któremu odpowidał. Przykładowo, jeśli pytającym jest „Truck #1”

Truck	Trailer
Truck #1	Trailer #1
Truck #0	Trailer #0
Truck #1	Trailer #15
Truck #8	Trailer #7
Truck #3	Trailer #9

Truck	Trailer
Truck #1	Trailer #1
Truck #1	Trailer #15

- d) W przeciwnym wypadku zapytanie jest buforowane, do momentu, gdy zostanie znalezione dopasowanie i będzie można utworzyć holon, wtedy zbuforowani agenci otrzymują odpowiedź „no”. W przypadku, gdy dopasowanie dotyczy zbuforowanego agenta, otrzymuje on odpowiedź „yes”
- e) W przypadku odpowiedzi „no” lista preferencji również ulega zmianie. W tym przypadku usuwane są wpisy dotyczące agenta, któremu odpowiadamy. Przykładowo, gdy odpowiadamy agentowi „Truck #1”:

Truck	Trailer
Truck #1	Trailer #1
Truck #0	Trailer #0
Truck #1	Trailer #15
Truck #8	Trailer #7
Truck #3	Trailer #9

Truck	Trailer
Truck #0	Trailer #0
Truck #8	Trailer #7
Truck #3	Trailer #9

- f) Przyjmowanie odpowiedzi na wcześniej wysłane zapytania – po przyjściu odpowiedzi sprawdzane jest czy jest ona pozytywna (“yes”) czy negatywna (“no”). Jeśli jest pozytywna to modyfikowana jest lista preferencji (jak w pkt. 3), a oba agenci są uzgodnieni. Wtedy odbywa się poszukiwanie kolejnego elementu, lub jeśli był to ostatni element wysyłana jest oferta do Dystrybutora.
- g) Jeżeli odpowiedź jest negatywna to lista preferencji modyfikowana jest jak w pkt. 5. Po modyfikacji rozpoczyna się poszukiwanie jak w pkt a.

Algorytm kończy działanie, kiedy lista preferencji jest pusta (wtedy wysyłana jest pusta oferta), lub gdy znalezione zostały wszystkie części holonu. Po nadaniu wszystkich ofert dystrybutor wylicza dla każdej z nich wartość funkcji kosztu i wybiera najlepszy. Następnie kieruje zapytania do Eunitów, o

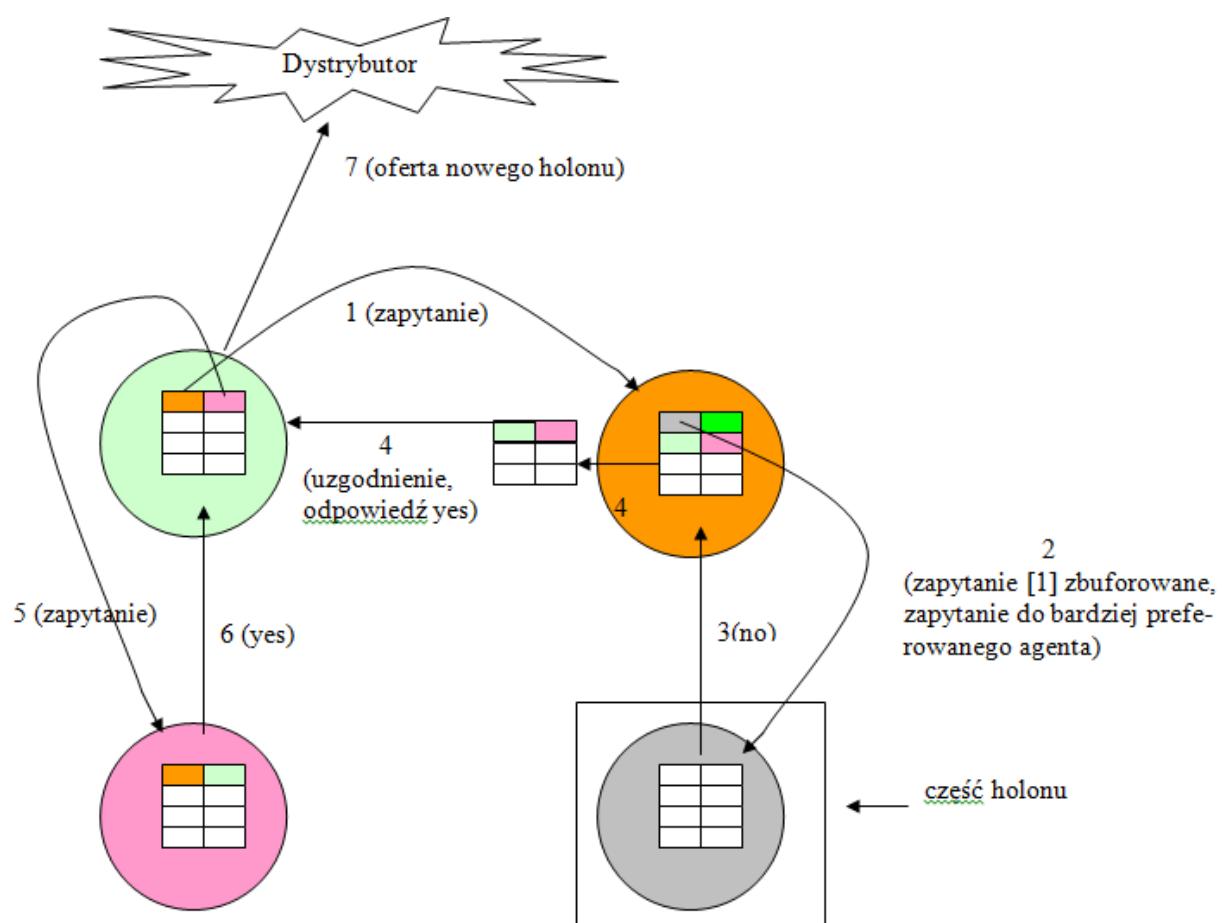
koszt ewentualnej obsługi zlecenia (Eunity stosują tą samą funkcję kosztu, co Dystrybutor!). Po zebraniu odpowiedzi wybierana jest ta o najlepszym koszcie.

W ostatniej fazie wybierana jest korzystniejsza z opcji i albo powstaje nowy holon, albo zlecenie jest przesyłane do Eunita.

W przypadku tworzenia nowego holonu:

- Informacje o składowych są przesyłane do pierwszego wolnego Eunita
- Każdy agent składowy dostaje powiadomienie, że jest częścią holonu. Informacja ta jest potem uwzględniana w procesie negocjacji. Obecnie agent po przyłączeniu się do holonu, nie może go opuścić, dlatego na wszelkie zapytania podczas negocjacji odpowiada „no”

**Przykład ilustrujący opisane wcześniej kroki:**



Rysunek 24 Przykład pokazujący różne etapy procesu negocjacji

### Zapobieganie zakleszczeniom

Łatwo widać, że przy przedstawionej wyżej procedurze bardzo łatwo dochodziłoby do zakleszczeń (przy różnych funkcjach kosztu).

Głównym czynnikiem powodującym takie zakleszczenia jest fakt, że każdy agent buforuje zapytania od agentów, których ma niżej w liście preferencji i próbuje „dogadać się” z agentami będącymi na szczycie listy (jeśli oni też są w podobnej sytuacji to mamy zapętlenie, z którego nigdy nie wyjdziemy).

Rozwiązaniem opisanej wyżej sytuacji było wprowadzenie trójwartościowej odpowiedzi na zapytanie o połączenie. Teraz agent może odpowiedzieć: „Tak”, „Nie” i „Nie wiem”.

Odpowiedź „Nie wiem” jest wysyłana, gdy pytana jednostka już kogoś pyta o połączenie.

Jeśli agent otrzymuje odpowiedź „Nie wiem”, to przesuwa pary z agentem, który mu w ten sposób odpowiedział na koniec listy preferencji.

Takie podejście rzeczywiście nie dopuszcza do zakleszczeń, ale jednocześnie wprowadza duży element losowości. Oznacza to, że po dwukrotnym jej uruchomieniu przy takich samych warunkach początkowych, możemytrzymać różne holony.

#### 3.4.2. Wymiana elementów

Z uwagi na to, że obecnie holon jest tworzony po otrzymaniu pojedynczego zlecenia i potem nie ulega zmianie, obsługa kolejnych zleceń jest nieoptimalna. Możliwe jest jednak rozszerzenie procesu negocjacji o część gdzie agent wchodzący w skład holonu mógłby go opuścić, gdyż byłoby to dla niego bardziej opłacalne.

Taki scenariusz mógłby wyglądać następująco:

- a) Wysłanie zlecenia
- b) Ustalenie list preferencji
- c) Wprowadzenie fazy wymiany – byłaby podobna w działaniu do wcześniejszej opisanej negocjacji, z tą różnicą, że inicjowałby ją agent, który chciałby się rozłączyć.

Wysyłałby zapytanie do wszystkich innych agentów wraz z zleceniami, które posiada już holon. Każdy agent mógłby podjąć decyzję o tym, czy chce dołączyć się do holonu. Decyzja byłaby podejmowana na podstawie wyliczonej wartości

funkcji kosztu i porównaniu jej z ewentualnym zyskiem utworzenia nowego holonu.

- d) Faza wymiany w pewnym momencie przechodziłaby w fazę negocjacji, ponieważ np. agent chciałby najpierw dowiedzieć się, z kim mógłby się połączyć, bo tylko wtedy znałby realną wartość funkcji kosztu dla danego zlecenia.

### 3.5. Realizacja wysyłania zleceń paczkami

System oferuje dwa tryby pracy. Pierwszy taki, jak opisany poprzednio, w którym dystrybutor wysyła kolejne zlecenia do agentów składowych. W tym przypadku negocjacje dotyczą tylko tego pojedynczego zlecenia i na podstawie ich wyniku tworzone są holony. Jak widać, wynik przy takim podejściu znacząco zależy np. od kolejności wysyłanych zleceń. Oczywiście tworzone holony również w większości przypadków nie są optymalne. Jest to zrozumiałe, ponieważ są tworzone w odpowiedzi na pojedyncze zlecenie. Kolejne zlecenia są im przyznawane na zasadzie, „jeśli możesz, to obsłuż”.

Alternatywą do takiego podejścia jest drugi tryb pracy, gdzie dystrybutor wysyła od razu wszystkie zlecenia. W tym przypadku odbywa się tylko jedna negocjacja. Po jej zakończeniu dystrybutor otrzymuje listę ofert do utworzenia nowych holonów.

#### 3.5.1. Przebieg tworzenia holonów w podejściu, gdzie zlecenia są wysyłane paczkami

- a) Dystrybutor wysyła wszystkie zlecenia do agentów
- b) Agenci tworzą listy preferencji. Wykorzystują do tego celu wiedzę o innych agentach, jak również informację o zleciennach
- c) Przeprowadzana jest negocjacja
- d) Dystrybutor otrzymuje oferty agentów i tworzy wszystkie możliwe holony – tworzenie holonów jest dynamiczne (nie trzeba podawać ich ilości w konfiguracji), a ich ilość jest ograniczona przez ilość agentów składowych
- e) Dystrybutor dla kolejnych zleceń pyta holony (Eunity) o koszt ich obsłużenia. Następnie wybiera najlepszą ofertę i przekazuje zlecenie
- f) W przypadku, gdy żaden holon nie może obsłużyć zlecenia symulowane jest wypożyczenie jednostki. Po prostu tworzony jest kolejny holon, który składa się z agentów podanych w pliku konfiguracyjnym (każdy wypożyczany holon jest taki

sam). Następnie holon ten normalnie bierze udział w procesie przyznawania zleceń

### 3.5.2. Algorytm tworzenia listy preferencji w przypadku wysyłania paczkami

W tym przypadku algorytm znaczco się różni, gdyż musi uwzględniać wszystkie zlecenia. Poniżej znajduje się uproszczony opis zastosowanego algorytmu, dla pojedynczego agenta:

- a) Inicjalizacja listy preferencji – agent na podstawie wiedzy o innych agentach tworzy listę zawierającą wszystkie możliwe kombinacje. Dodatkowo każda para w liście posiada swoją niezależną listę zleceń, które holon z niej powstały mógłby zrealizować

#### UWAGA!

Należy zwrócić uwagę, że nie każdy agenci mogą się ze sobą łączyć. Np. połączenie może zależeć od typu połączenia. Czynniki te zostały uwzględnione w algorytmie

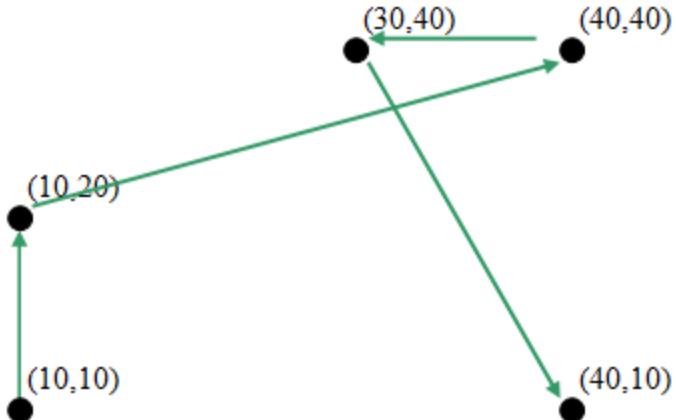
- b) Dla kolejnego zlecenia sprawdzane jest, czy przyszły holon mógłby zrealizować dane zlecenie – odbywa się to w oparciu o listę zleceń danej pary.
- c) W przypadku, gdy zlecenie może zostać zrealizowane, jest dodawane do lokalnej listy danej pary
- d) Jeśli nie może być zrealizowane, sprawdzane jest, czy zamiana któregoś zlecenia z lokalnej listy danej pary ma lepszą wartość funkcji kosztu. Wybór realizowany jest tak, że każde zlecenie z listy jest zamieniane z obecnie rozpatrywanym zleceniem i wtedy wyliczane są funkcje kosztu. Jeśli któraś z konfiguracji ma lepszą wartość funkcji kosztu, to modyfikowana jest lokalna lista danej pary. Do wyliczania kosztu jest brany łączny dystans zleceń w liście, oraz ich łączna ładowność.

#### UWAGA!

Dystans nie jest wyznaczany optymalnie. Zastosowano tutaj bardzo prosty algorytm. Na dystans składają się odległości między poborem, a dostawą dla każdego zlecenia, jak i dystans między miejscem dostawy wcześniejszego zlecenia z listy i miejscem poboru późniejszego zlecenia. np.: dla listy zleceń

PICKUP	DELIVERY
(10,10)	(10,20)
(10,20)	(40,40)
(30,40)	(40,10)

Dystans liczony jest następująco:



$$[(10-10)^2 + (20-10)^2] + [(10-40)^2 + (20-40)^2] + [(30-40)^2 + (40-40)^2] + [(30-40)^2 + (40-10)^2]$$

- e) Po rozpatrzeniu wszystkich zleceń, lista preferencji jest sortowana. Pierwsze są pary, które mogą zrealizować największą ilość zleceń. Jeśli jest kilka takich par, to za kryterium przyjęta jest wartość funkcji kosztu. W ten sposób pierwsze są konfiguracje, które mogą zrealizować najwięcej zleceń, przy najmniejszym koszcie.

### 3.5.3. Częste problemy

Podczas używania trybu wysyłania paczkami często zdarza się, że JVM ma za mało pamięci. Pamięć ta jest przyznawana podczas inicjalizacji JVM, dlatego niezależnie od tego ile mamy pamięci RAM możemy dostać błąd podczas symulacji.

Jeśli JVM zwróci błąd podczas symulacji, mamy dwa wyjścia:

- a) Uruchomić Eclipse'a z następującymi parametrami:

W systemie Linux:

a. `eclipse -vmargs -Xms256m -Xmx1024m`

lub pod Windowsem:

b. `eclipse.exe -vmargs -Xms256m -Xmx1024m`

- b) Zmienić zawartość pliku `eclipse.ini`. Należy zmienić dwie ostatnie linijki:

- a.  $-Xms40m$  na  $-Xms256m$
- b.  $-Xmx256m$  na  $-Xmx1024m$

### 3.6. Funkcje kosztu

Dodatkowo system umożliwia definiowanie różnych funkcji kosztu dla agentów [rozdział 9]. Każdy agent (nawet tego samego rodzaju) może korzystać z innej funkcji kosztu. Funkcję podajemy jako string. Szczegółowy opis tworzenia funkcji znajduje się w rozdziale dotyczącym konfiguracji [rozdział 8].

W większości symulacji została użyta funkcja taka jak w poprzednim systemie, czyli:

$$0.01 * dist * (4 - comfort) + (dist / 100) * fuel * ((mass + load) / power)$$

natomiast obecnie stosujemy funkcję:

$$dist$$

Oprócz agentów składowych funkcje kosztu znajdują się również u Dystrybutora i Eunita (takie same). Niestety w obu przypadkach funkcje te nie są konfigurowalne, aby je zmienić należy wprowadzić zmiany w kodzie:

- Dla Dystrybutora – `dtp.jade.distributor.DistributorAgent`, metoda:  
`chooseBestHolon`
- Dla EUnitów – `dtp.jade.eunit.ExecutionUnit`, metoda `getRatio`

Funkcja kosztu ustawiana dla TransportAgent'ów jest używana w dwóch miejscach:

- Przy procesie negocjacji
- Przy obliczaniu kosztu realizacji nowego zlecenia, przez holon'a – brana jest suma funkcji kosztu jego składowych

### 3.7. Tworzenie jednostek transportowych – Eunity

Eunity są tworzone dynamicznie, w trakcie symulacji. Jest to lepsze podejście niż ustalanie statycznej ich ilości, ponieważ mają za zadanie jedynie reprezentować składowe holonu. Nie biorą natomiast udziału w procesie wyboru składowych. Takie podejście umożliwia również wprowadzenia symulacji wypożyczania jednostek z zewnętrz i obliczania płynących z tego ewentualnych zysków i strat.

Obecnie możliwa jest konfiguracja tylko jednego typu wypożyczanego EUnita. Odpowiednie parametry należy podać w pliku konfiguracyjnym. Domyślny EUnit jest tworzony w przypadku, gdy dane zlecenie nie może zostać obsłużone przez istniejące jednostki. Po stworzeniu, domyślny EUnit jest traktowany tak samo jak inne Eunity. Po

prostu przyjęto, że płacimy, za samo wypożyczenie, a nie np. dystans jaki pokona taka jednostka. W innym podejściu, gdzie zależy nam na wykorzystywaniu wypożyczonej jednostki jak najrzadziej, można wprowadzić pewien podział przy rozdzielaniu zleceń, np.:

- Najpierw sprawdzić, czy nasza jednostka może obsłużyć zlecenie
- W przypadku gdy może – przekazać jej zlecenie i nie pytać o oferty domyślnych Eunitów
- Jeśli nie może – wtedy spytać domyślne EUnity
- Jeśli któryś może wykonać zlecenie, to mu je przekazujemy, jeśli nie to dopiero wtedy powstaje nowy domyślny Eunit

## UWAGA

Ponieważ konfiguracja domyślnych EUnitów jest statycznie wpisana w konfiguracji, może się zdarzyć, że zlecenie nie może być obsłużone nawet przez nowego (pustego) domyślnego Eunita. W takim przypadku jest wypisywany odpowiedni komunikat i system kończy działanie!

## 3.8. Rozdział zleceń przez Dystrybutora

System oferuje dwa tryby rozsyłania zleceń:

### 3.8.1. Preferowany koszt

W tym trybie decyzja o tym, czy zlecenie trafia do istniejącego Eunita, czy może tworzymy nowego.

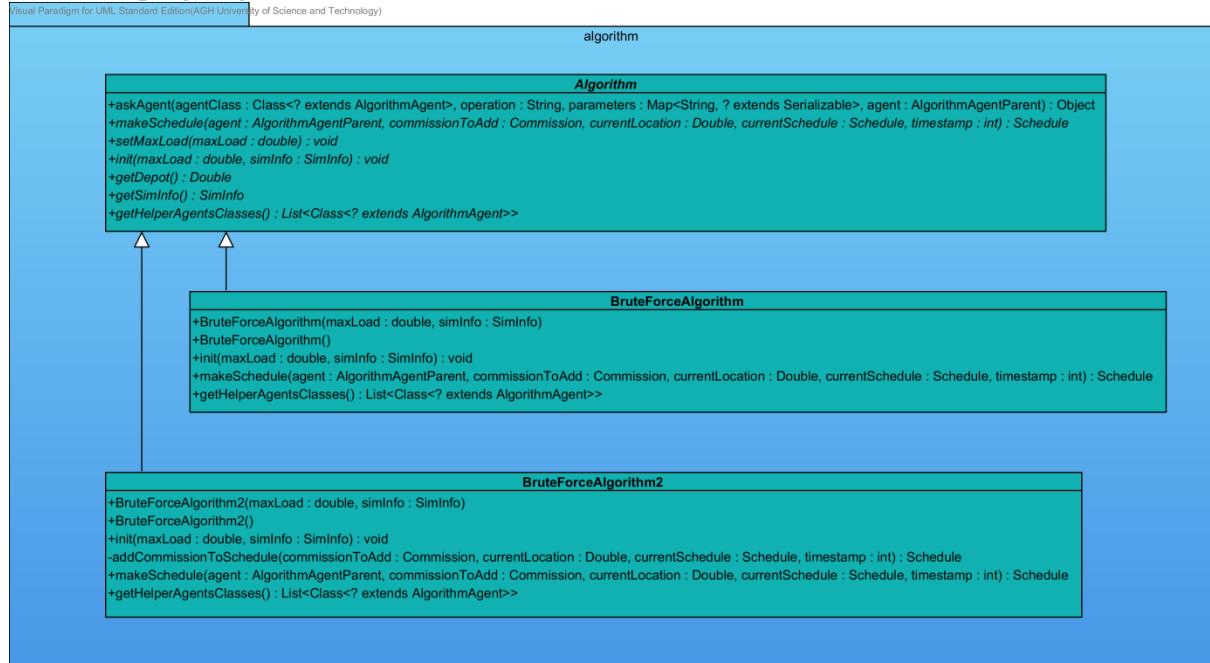
### 3.8.2. Preferowane wykorzystanie

Tutaj chcemy załadować Eunita w jak największym stopniu. Działanie w tym trybie polega na tym, że jeżeli jakieś istniejące Eunity mogą zrealizować zlecenie, to wybierany jest ten o najlepszym koszcie i wysyłane jest do niego to zlecenie (nawet jeśli ewentualny koszt nowego Eunita byłby lepszy).

## 3.9. Zaimplementowane algorytmy

Poniższy rozdział zawiera opis zaimplementowanych algorytmów. Każdy z nich zawiera zarówno opis słowny, jak i pseudokod służący ułatwieniu zrozumienia ich działania.

### 3.9.1. Algorytmy wstawiania zleceń



Rysunek 25 Diagram klas przedstawiający algorytmy wstawiania zleceń

#### a) Algorytm pełnego przeglądu – BruteForce

Algorytm ten stosuje pełny przegląd możliwości wstawiania zleceń, dlatego można utożsamiać go z algorytmem bruteforce, dla tego problemu (oczywiście w ramach jednostki)

#### Opis słowny

W tym algorytmie jednocześnie wstawiane są oba zlecenia (zarówno pickup, jak i delivery). Kolejność zleceń do dodania nie ma znaczenia, ponieważ mogą one być wstawiane w dowolnym miejscu. Dodanie pary zleceń odbywa się w następujący sposób:

1. Wstawiane jest zlecenie pickup w każde możliwe miejsce kalendarza
2. Wstawiane jest zlecenie delivery w każde miejsce będące po wcześniejszej wstawionym zleceniu pickup
3. Wyliczany jest koszt każdej konfiguracji
4. Wybierana jest najlepsza

W ten sposób rozdzielone zostają wszystkie zlecenia z listy. Oczywiście w trakcie wstawiania sprawdzane są ograniczenia czasowe i pojemościowe. Jeśli w żaden sposób nie da się ich zaspokoić to zwracana jest informacja o tym, że nie da się zrealizować zleceń.

### Obsługa problemu dynamicznego

Wprowadzenie problemu dynamicznego spowodowało powstanie dodatkowych ograniczeń, którymi są:

- zestaw zleceń w kalendarzu został podzielony na takie, które są już zrealizowane (lub w trakcie realizacji) i pozostałe przydzielone jednostce. Pierwsza grupa jest specyficzna, gdyż nie można jej już modyfikować
- kalendarz danej jednostki może się zmienić dopiero od zlecenia które jest kolejne do zrealizowania. Oznacza to, że nie dopusczamy sytuacji, gdzie jednostka jedzie do jakiegoś punktu i w trakcie zmienia swój cel zjeżdża zbranej trasy)

Jak widać przedstawione wyżej problemu sprawiły że w algorytmie należy rozróżnić dwa rodzaje zleceń (zrealizowane i nie). Jest to realizowane w taki sposób, że w samym algorytmie lista zleceń do której wstawiamy nowe zlecenie jest niejako obcinana i brana pod uwagę jest tylko część która może być modyfikowana.

### Pseudokod

```
/*
 * commissionToAdd – zlecenie które ma być dodane do kalendarza
 * currentLocation – bierzące położenie jednostki – obecnie nieużywane (null)
 * currentSchedule – bieżący kalendarz do którego chcemy wstawić zlecenie
 * timestamp – znacznik czasowy przebiegu symulacji
 */
Schedule makeSchedule(commissionToAdd, currentLocation, currentSchedule, timestamp)
begin
    schedule=currentSchedule
    load=0.0;
    bestIndex=-1;
    bestIndex2=-1;
    /* Ta metoda zwraca index pierwszego zlecenia które możemy modyfikować */
```

```

begin = currentSchedule.getNextLocationId(depot, timestamp)
for(i=begin;i<schedule.size();i++)
begin
    dodanie commissionToAdd na pozycje i jako pickup
    for(j=i+1;j<schedule.size();j++)
    begin
        dodanie commissionToAdd na pozycję j jako delivery
        if(schedule jest lepszy od najlepszego dotychczas i spełnia ograniczenia
czasowe i ładowności)
            begin
                bestIndex=i
                bestIndex2=j
            end
            usuń commissionToAdd z pozycji j
        end
        dodaj commissionToAdd jako delivery na koniec
        if(schedule jest lepszy od najlepszego dotychczas i spełnia ograniczenia czasowe i
ładowności)
            begin
                bestIndex=i;
                bestIndex2=-2;
            end
            usuń zlecenie commissionToAdd z końca (delivery), oraz zlecenie z pozycji i
(pickup)
        end
    end
    dodaj commissionToAdd jako pickup, a potem delivery na końcu
    if(schedule jest lepszy od najlepszego dotychczas)
    begin
        return schedule
    end
    usuń dwa ostatnie zlecenia
    if(bestIndex== -1 or bestIndex2== -1) return null;
    wstaw commissionToAdd jako pickup na pozycje bestIndex
    if(bestIndex2== -2) wstaw commissionToAdd jako delivery na koniec
    else wstaw commissionToAdd jako delivery na pozycji bestIndex2

    return schedule;
end

```

b) BruteForce2 – odmiana algorytmu 3.8.1

Algorytm BruteForce2 wykorzystuje algorytm BruteFoce. Różnica polega na tym, że w algorytmie BruteForce jest wstawiane tylko jedno zlecenie, natomiast w algorytmie BruteForce2 gdy chcemy wstawić nowe zlecenie wszystkie zlecenia (których kolejność możemy zmieniać) są usuwane z kalendarza i dodawane do listy. Do tej listy jest

również dodawane nowe zlecenie. Następnie wszystkie zlecenia z listy kolejno są dodawane do kalendarza.

Na pierwszy rzut oka mogłoby się wydawać, że gdybyśmy użyli algorytmu BruteForce to wynik byłby ten sam. Oczywiście jest w tym trochę racji, ale nie do końca. Należy zwrócić uwagę na to, że zlecenia są dokładane za pomocą SimulatedTrading a także kolejność ich wstawiania również nie jest bez znaczenia (bo np. jakiś wcześniejsze zlecenie zostanie wstawione tak, że kolejne nie może już być wstawione z uwagi na niespełnienie ograniczeń)

### 3.9.2. Dodatkowe klasy i metody

- *Helper*

Klasa pomocnicza. Posiada jedynie metody statyczne. Zawiera metody, które występują w aplikacji, a są związane z agentami (np. ExecutiveUnitAgent'em). Z uwagi na to, że nie opłaca się tworzyć tych agentów i na nich wołać te metody (co nie byłoby również prostą rzeczą) postanowiłem skopiować ich implementację (z niezbędnymi zmianami) do klasy Helpera. Obecnie ma on metody:

- getRatio – oblicza koszt realizacji zlecenia w przypadku, gdy wszystkie jednostki są takie same.
- calculateDistance – oblicza dystans między dwoma lokalizacjami
- calculateCost – oblicza koszt realizacji wszystkich zleceń z kalendarza
- calculateCalendarCost – oblicza sumę kosztów realizacji zleceń wszystkich zleceń
- copy – kopiuje mapę z holonami

- *Schedule*

Nowa klasa reprezentująca w prosty (i zrozumiały) sposób kalendarz. Jej głównym elementem są listy

- commissions – zawierająca kolejne zlecenia (pickup i delivery)
- types – określa rodzaj zlecenia na odpowiadającej pozycji w commissions

Dodatkowo klasa dostarcza szeregu metod do dodawania i usuwania zleceń, jak również kilku metod pomocniczych, jak np. metodę obliczającą przebyty dystans

(po ułożeniu kalendarza), lub zwracającą lokalizację ostatnio dodanego zlecenia.

Ogólnie o klasie Schedule można myśleć jak o kontenerze zleceń, który trzyma je w pewnym porządku.

### **3.9.3. Simulated trading**

Rozdział zawiera opis zaimplementowanego algorytmu simulated trading. Jest to ten sam algorytm, co w starym systemie ze zmianami niezbędnymi do przystosowania go do nowego systemu.

#### **3.9.3.1. Idea**

Algorytm simulated trading, jak sama nazwa mówi, ma za zadanie symulowanie procesu licytacji. W naszym systemie licytacji podlegają zlecenia. Dzięki algorytmowi, możliwe jest wymienianie zleceń między poszczególnymi Eunitami, w taki sposób, żeby zminimalizować koszt ich obsługi. Proces licytacji jest uruchamiany podczas próby przyznania nowego zlecenia do jednostki transportowej. Ogólny schemat w tym przypadku wygląda następująco:

- proces wyboru najlepszego kandydata, który powinien otrzymać zlecenie do realizacji
- wysłanie zlecenia do tego Eunita
- rozpoczęcie simulated trading
- każdy Eunit zaczyna licytować swoje najgorsze zlecenie – jako najgorsze zlecenie przyjmowane jest to, które w największym stopniu zwiększa pokonywany dystans. Zlecenie to jest usuwane z kalendarza i przekazywane na licytację. Ponadto tworzona jest kopia obecnego kalendarza (przed usunięciem zlecenia) – służy ona do odtworzenia kalendarza w przypadku, gdy żaden Eunit nie chce tego zlecenia i wraca ono do nadawcy.
- wybór zwycięzcy licytacji jest dokonywany podobnie jak przydziął nowego zlecenia, czyli każdy Eunit wysyła swoją ofertę do Dystrybutora, a ten wybiera najlepszą z nich
- ostatnią fazą licytacji jest powiadomienie zwycięzcy (poprzez przesłanie mu zlecenia)

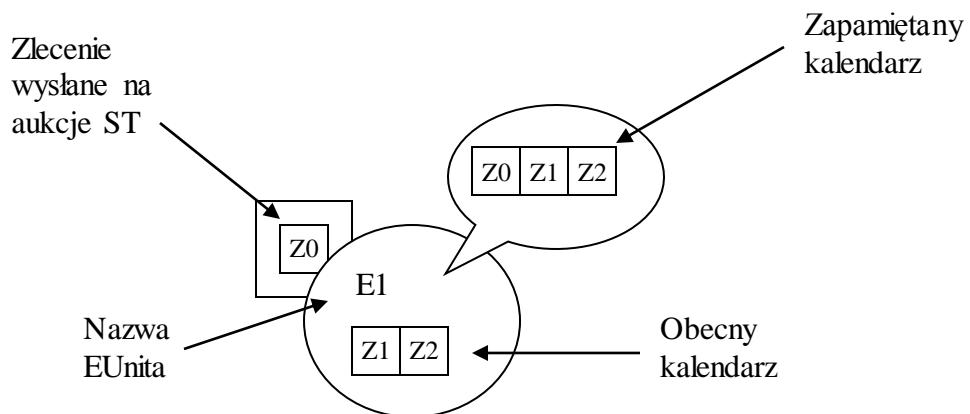
- Eunit, który wygrywa licytację ma dwa scenariusze działania. Jeśli zwycięzca wysłał to zlecenie to odtwarza kalendarz, jeśli nie to po prostu dodaje to zlecenie do swojego kalendarza

Poniżej przedstawiony jest przykład obrazujący pojedyncze uruchomienie algorytmu simulated trading:

Identyfikatory:

- EX – oznacza Eunita o numerze X
- ZX – oznacza zlecenie o numerze X

Przyjęte oznaczenia:



Sytuacja początkowa:

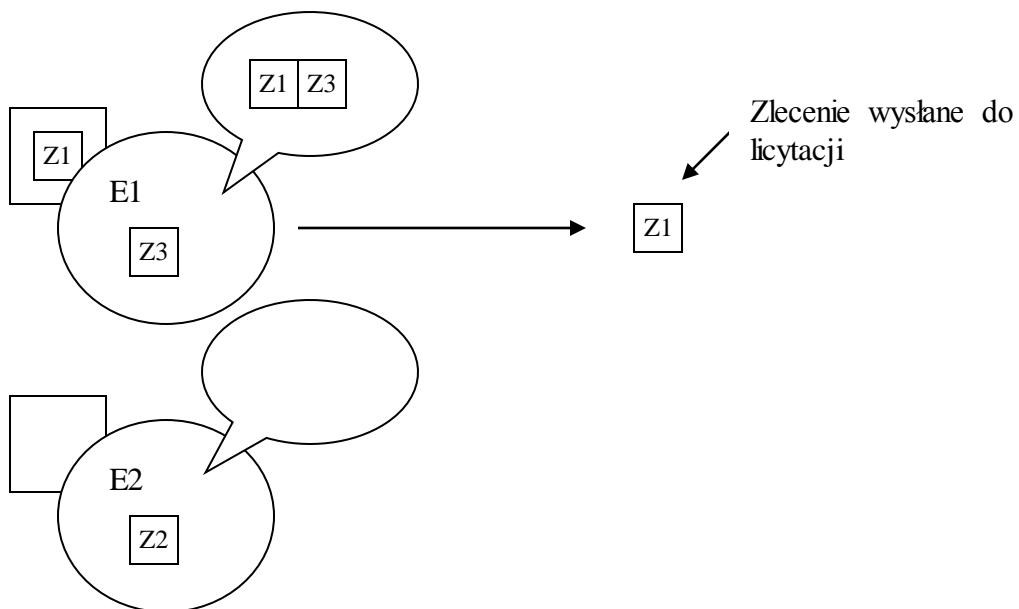
Mamy dwa Eunity:

- E1 posiadający zlecenie Z1
- E2 posiadający zlecenie Z2

Przychodzi nowe zlecenie Z3 i trafia do E1. Rozpoczyna się simulated trading:

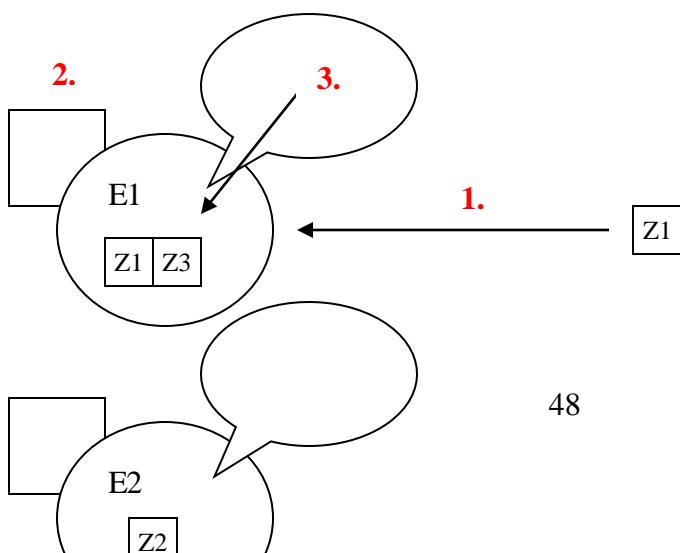
- I. E1 wybiera swoje najgorsze zlecenie (jest nim Z1 - zlecenie powodujące największy przyrost dystansu). Następuje zapamiętanie Z1 w lokalnej zmiennej

E1, usunięcie go Z1 z kalendarza, wykonanie kopii jego kalendarza i przesłanie zlecenia na licytację.

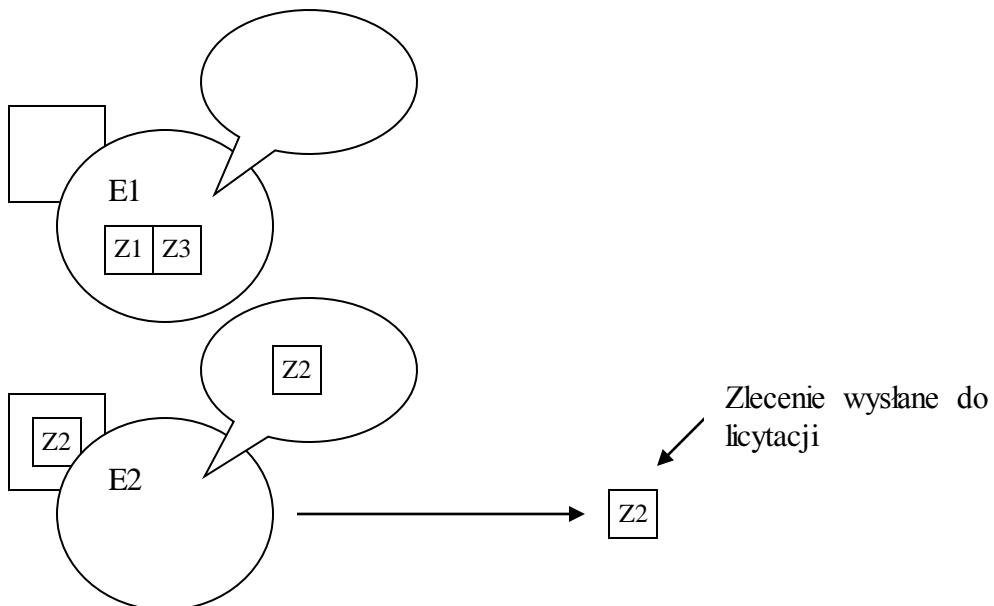


II. Z1 jest licytowane jak każde inne zlecenie. Powiedzmy że licytacje wygrywa E1. W tej sytuacji:

1. Z1 wraca do niego
2. Następuje porównanie z zapamiętanym zleceniem. Na tej podstawie E1 stwierdza, że to on wcześniej wysłał Z1 do licytacji. W tym przypadku usuwa Z1 ze zmiennej, w której jest ono zapamiętane
3. przywraca kalendarz z backup'u (i go kasuje)

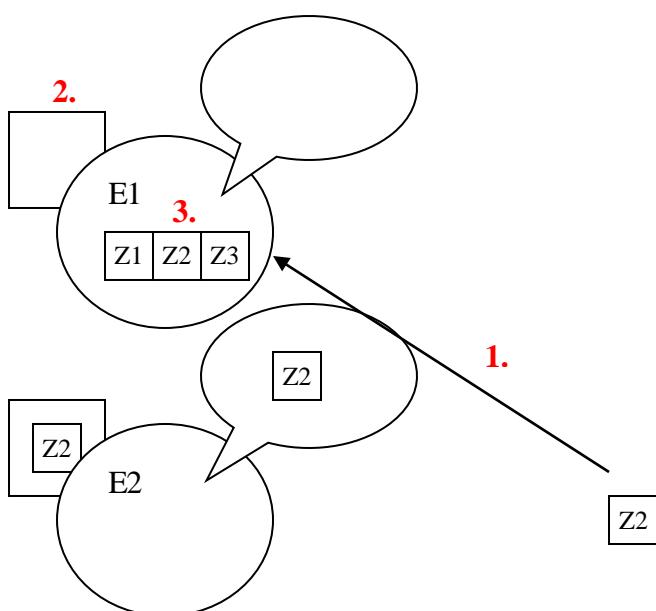


III. Teraz swoje najgorsze zlecenie wysyła E2 (analogicznie jak E1 w punkcie I)



IV. Znowu odbywa się licytacja, tym razem zlecenia Z2. Wygrywa ją E1. W związku z tym realizowane są kroki:

1. Z2 trafia do E1
2. E1 sprawdza, czy zlecenie które otrzymał jest takie samo jak zapamiętane zlecenie (czy jest to zlecenie które wcześniej wysłał). Jak widać E1 nie ma zapamiętanego żadnego zlecenia, dlatego wynik porównania `null==Z2` wskazuje, że należy zwyczajnie dodać zlecenie do kalendarza.
3. Z2 zostaje dodane do aktualnego kalendarza



### 3.8.3.2. *SimulatedTrading*

#### Schemat działania fullSimulatedTrading i jego pseudokod

Działanie algorytmu jest bardzo proste. Po dodaniu zlecenia holon inicjuje procedurę „*Simulated Trading*”. Wybiera swoje najgorsze zlecenie i wysyła je do licytacji. Teraz zlecenie to jest traktowane jak każde inne zlecenie. Każdy holon wysyła swoją ofertę (koszt dodania zlecenia). Wybierany jest holon, który wysyła najlepszą ofertę. W przypadku, gdy jest to inny holon (nie ten który wysłał to zlecenie do licytacji), to teraz on inicjuje *Simulated Trading*. Gdy wysłane na licytację zlecenie wraca do nadawcy, to wtedy wysyłane jest kolejne „najgorsze zlecenie”. Procedura kończy się, nie ma już zleceń do wysłania (wartość zmiennej STDepth jest większa od ilości zleceń w kalendarzu).

#### Pseudokod

```
/* Przekazywane parametry:
 * holons – mapa której kluczami są identyfikatory holonów, a wartościami ich kalendarze
 * (Schedule). Reprezencja holonów w symulacji
 * holon – jest to klucz z mapy holons aktualnego holonu (inicjującego ST)
 * STDepth – określa które najgorsze zlecenie ma być brane pod uwagę
 * algorithm – algorytm przydziału zleceń w obrębie holonu
 * commissionsId – zbiór identyfikatorów zleceń, które były już wysyłane na aukcję w
 * ramach
 * ST (służy zapobieżeniu zapętlenia algorytmu)
 * time – określa czy najgorsze zlecenie ma być wybierane po czasie bezczynności
 * (time=false), czy ogólnym czasie dostarczenia wszystkich zleceń (time=true)
 * timestamp – obecny znacznik czasowy symulacji
 */
Map<AID, Schedule> fullSimulatedTrading(holons, holon, STDepth, depot, algorithm,
commissionsId, time, timestamp)
begin
    schedule = holons.get(holon)
    backup = utwórz kopię bieżącego kalendarza (schedule)
    worstCommission = pobierz najgorsze zlecenie z bieżącego kalendarza
    (uwzględniając
        zmienną time, oraz STDepth)
    /* worstCommission jest null'em, gdy STDepth jest za duże (większe od ilości zleceń
     * w kalendarzu)*/
    if(worstCommission==null) then
        begin
            holons.put(holon,backup);
            return holons;
```

```

end
if(commissionsId.contains(worstCommission.getID())) then
begin
    return holons;
end
commissionsId.add(worstCommission.getID());
for każdy holon (h) do
begin
    schedule = pobierz kalendarz holonu h
    if worsCommission może być dodane do kalendarza schedule then
begin
        extraDistance=(dystans po dodaniu zlecenia) – (dystans przed
                     dodaniem)
        if(bestCost>getRatio(extraDistance,worstCommission) then
        begin
            bestCost=getRatio(extraDistance,worstCommission);
            bestHolon=h
            bestSchedule=schedule
        end
        added=true;
    end
end
if(added==false) then
begin
    holons.put(holon,backup);
    return holons;
end

holons.put(bestHolon,bestSchedule)
if(bestHolon==holon) return fullSimulatedTrading(holons, holon, STDepth+1,
                                                 depot, algorithm, commissionsId, time)
else return fullSimulatedTrading(holons, bestHolon, 1, depot, algorithm,
                                 commissionsId, time)
end

```

### Schemat działania complexSimulatedTrading i jego pseudokod

Idea algorytmu jest bardzo prosta, jeśli chodzi o założenia, natomiast dość skomplikowana w realizacji. Należało bowiem zapewnić jakiś mechanizm deklaracji przyjęcia i oddania zleceń przez jednostki. W wielu źródłach jest to realizowane poprzez graf przydziału. Który potem służy do wybrania odpowiednich zamian. Tutaj jednak zastosowane zostało inne podejście, ze względu na to, że nasza aplikacja jest oparta na agentach i to oni powinni decydować o zamianach (a nie jakiś scentralizowany punkt, gdzie byłby tworzony graf i wybierane rozwiązanie). Algorytm składa się z kilku faz.

Najpierw typowane są zlecenia do zamiany (w odpowiedzi na pojawienie się jakiegoś zlecenia, którego nie jesteśmy w stanie zrealizować). Następnie nowe zlecenie jest dodawane do kalendarza, a wysyłane są kolejno zlecenia do zamiany. Jeśli jakiś holon przyjmie takie zlecenie to zamiana się dokonuje, jeśli żaden nie będzie w stanie tego zrobić, to zlecenie jest odrzucane. Należy tutaj wspomnieć, że proces wymian jest bardziej złożony, gdyż każde zlecenie, które nie może zostać dodane do kalendarza jest uruchamia proces negocjacji wymian.

### Pseudokod

```

/* Metoda zwracająca listę zleceń, z których jedno musi być zrealizowane przez innego holona, żeby można było obsłużyć zlecenie commission
 * commission – zlecenie, które chcemy dodać
 * schedule – kalendarz holona, do którego chcemy dodać zlecenie commission
 * algorithm – algorytm służący do rozdziału zleceń
 */
List getCommissionsToReplace(commission, schedule, algorithm)
begin
    backup = stwórz kopię schedule
    for każde zlecenie (com) ze schedule, które może zostać zmienione do
    begin
        schedule.removeCommission(com)
        if można dodać commission then
            begin
                result.add(new Container(dystans jaki musi przejechać jednostka po
                                zmianie kalendarza, com))
            end
        end
    end
    /* sortowanie po dystansie malejąco*/
    sort(result)
    return result
end

/* Metoda symulująca złożone wymiany między jednostkami
 * holons – mapa której kluczami są identyfikatory holonów, a wartościami ich kalendarze
 * (Schedule). Reprezencja holonów w symulacji
 * com – zlecenie, które chcemy wstawić
 * algorithm – algorytm szeregowania zleceń
 * depth – określa jak wiele wymian może być dokonanych w ramach dodawania jednego
 * zlecenia
 * comsId – zbiór identyfikatorów zleceń, które były wymieniane
 * timestamp – znacznik czasowy symulacji
 */
Map<AID, Schedule> complexSimulatedTrading(holons, com, algorithm, depth, comsId,

```

```

timestamp)
begin
    if(depth==0) return null
    holonsBackup = kopia mapy holons
    for(każdy holon h) do
        begin
            holons = kopia holonsBackup
            schedule = kalendarz holonu h
            scheduleBackup = kopia schedule
            if( można dodać com do schedule) then
                begin
                    holons.put(h,nowy schedule)
                    return holons
                end
            commissions = getCommissionsToReplace(com, schedule, algorithm)
            for(każdy element (comToReplace) listy commissions) do
                begin
                    if(comsId.contains(comToReplace)) continue
                    else comsId.add(comToReplace)
                    if(com.getID()==comToReplace.getID()) continue
                    holons = kopia holonsBackup
                    schedule = kopia shceduleBackup
                    schedule.removeCommission(comToReplace)
                    dodaj com do schedule (algorithm.makeSchedule)
                    holons.put(h, nowy schedule)
                    holonsTmp=complexSimulatedTrading(holons, comToReplace,
                                                     algorithm, depth-1,comsId)
                    if(holonsTmp!=null) return holonsTmp
                end
            end
            return null
        end
    end

```

### 3.8.3.3. Wybór „worstCommission”

Wybór „worstCommission” następuje w kilku krokach:

1. tworzona jest lista zawierająca pary (zlecenie, koszt)
2. lista ta jest uzupełniana (koszt jest kalkulowany na dwa sposoby, o czym potem)
3. następuje sortowanie listy po kosztach (od największego do najniższego)
4. wybierane jest zlecenie zgodnie z wartością STDepth

Jak widać o wyborze „najgorszego zlecenia” decyduje wyliczanie kosztu danego zlecenia.

Obecnie mamy cztery metody jego obliczania:

1. czas obsługi wszystkich zleceń – kosztem jest czas realizacji wszystkich zleceń, po usunięciu zlecenia, będącego kandydatem na worstCommission
2. czas oczekiwania – tutaj brany pod uwagę jest czas oczekiwania jednostki na realizację każdego zlecenia (jest to suma czasu oczekiwania na pickup i delivery)
3. całkowity czas obsługi zleceń z uwzględnieniem kary
4. całkowity dystans potrzebny na realizację zleceń z uwzględnieniem kary

Oczywiście podczas dokonywania wyboru brane są pod uwagę tylko zlecenia, które nie zostały jeszcze obsłużone (i nie są w realizacji)

#### **3.8.3.4. Uruchamianie algorytmów ST**

Jak widać mamy dwa różne algorytmy Simulated Trading. Nie są one jednak uruchamiane w tym samym czasie (przy tych samych warunkach). FullSimulatedTrading jest uruchamiane, jeśli jakaś jednostka otrzymała nowe zlecenie. W przypadku, gdy żaden holon nie może przyjąć nowego zlecenia, normalnie byłaby tworzona nowa jednostka. Mimo to wcześniej uruchamiany jest algorytm complexSimulatedTrading, który próbuje tak przeorganizować kalendarze poszczególnych jednostek, żeby dało się zrealizować zlecenie bez konieczności tworzenia dodatkowego holonu.

Oczywiście wyżej opisane postępowanie jest prawdziwe, jeśli algorytmy simulated trading są aktywne. W konfiguracji możemy całkowicie wyłączyć ST, lub ustawić częstotliwość jego uruchamiania (i wtedy tylko w skonfigurowanych warunkach będzie mogło być uruchomione).

Obecnie mamy do wyboru ustawienie częstotliwości uruchamiania algorytmów ST na podstawie:

- ilości zleceń – algorytmy ST będą mogły być uruchomione co n zleceń
- ilości czasu – w tym przypadku ustawiamy co ile timestampów ST może być uruchomione

#### **UWAGA:**

Ustawianie częstotliwości dla algorytmów ST działa następująco (powiedzmy, że wybrano STTimeGap = 4). Co 4 timestampy ST może być uruchomione, ale oczywiście nie oznacza to,

że zostanie uruchomione. Jeśli w danym timestampie nie musiało być uruchamiane ST to zostanie uruchomione w najbliższym kolejnym. Przykładowo, jeśli mamy STTimeGap = 4 i zlecenia napływają w następujących znacznikach czasowych: 0, 2, 3, 4, 7, 10, 12, 15, 20..., to ST będzie uruchomiony w: 0, 4, 10, 12, 15, 20

### 3.8.3.5. Opis konfiguracji algorytmów

Obecnie jeśli chodzi o konfigurację algorytmów w systemie możemy ustawić:

- Wybór „worstCommission” – może ono być wybierane po największym czasie oczekiwania (waitTime), lub jest to zlecenie, które wprowadza największy przyrost czasu realizacji zleceń – worstCommissionByGlobalTime (true/false)
- Stopień zagłębiania w algorytmie complexSimulatedTrading – jest to parametr określający ile maksymalnie może zostać wykonanych zamian typu „wezmę to zlecenie, jeśli ktoś weźmie ode mnie inne zlecenie” – STDepth (int)
- Algorytm szeregowania zleceń w ramach każdego eunita – algorithm (nazwa algorytmu)
- Sposób wyliczania kosztu nowego zlecenia – kosztem może być czas realizacji, lub koszt jest wyliczany na podstawie funkcji kosztów agentów składowych – dist (true/false)
- Częstość uruchamiania algorytmów ST – po ilości obsłużonych zleceń, lub po ilości znaczników czasowych – STCommissionGap lub STTimeGap (int)
- Oczywiście wcześniejsze parametry dotyczące tworzenia samych holonów zostały zachowane

### 3.8.3.6. Porównanie algorytmu Simulated Trading zaimplementowanego w systemie z wersją klasyczną

Porównanie dotyczy algorytmu przedstawionego w: *The simulated trading heuristic for solving vehicle routing problems – A. Bachem, W. Hochstättler, M. Malich*

Na początku przedstawiona zostanie koncepcja i zasada działania podanego wyżej algorytmu, a potem zostanie ona porównana z wersją zaimplenetowaną w systemie.

#### 3.8.3.6.1. Wersja klasyczna

Główna pętla algorytmu wygląda tak:

```
do
  Sell-And-Buy phase
  Trading-Matching-Search phase
```

```
if Trading-Matching M is found then
    Update tourplan T according to M
while timelimit is reached
```

Jak widać jest to algorytm wieloprzebiegowy, polegający na budowie grafu, który służy do sprawdzania, czy są możliwe skomplikowane, wielokrotne wymiany między jednostkami w celu poprawy rozwiązania. Poniżej są pobiędnie przedstawione kolejne fazy.

### Sell-And-Buy

Faza polega na konstrukcji grafu – jego wielkość jest ograniczona przez odpowiedni parametr. Ogólnie jest to kolejna pętla przechodząca po planach wszystkich pojazdów i dla każdego z nich jest losowana z pewnym prawdopodobieństwem akcja:

- Sprzedaż
  1. Wybierane jest zlecenie, które powoduje największy przyrost kosztu
  2. Zlecenie jest usuwane z planu i dodawany jest odpowiedni węzeł do grafu (sell)
- Kupno
  1. Dodanie węzła do grafu (buy)
  2. Dodanie odpowiednich krawędzi wychodzących z wierzchołków sell, które dotyczą tego samego zlecenia

Wagami w powstałym grafie dwudzielnym jest polepszenie kosztu (dodatnie/ujemne).

### Trading-Matching

Poszukujemy takiego podzbioru krawędzi, które dają globalnie najniższy koszt. Dopasowanie musi być takie, żeby każda krawędź wychodziła z węzła (sell), z którego nie wychodzą żadne inne dopasowane krawędzie i wchodzić do węzła (buy), do którego nie wchodzą żadne inne dopasowane krawędzie. Niestety problem znalezienia takich dopasowań jest NP.-zupełny. Modyfikujemy trasy zgodnie ze

znalezionym dopasowaniem. Musimy tutaj również pamiętać, że nie wystarczy brać pod uwagę kosztu, ale również poprawność tworzonych tras (np. ograniczenia czasowe).

### 3.8.3.6.2. Porównanie z algorytmem zaimplementowanym w systemie

Z dwóch algorytów ST, zaimplementowanych w systemie najbardziej podobny jest ComplexSimulatedTrading i to z nim porównywany jest algorytm klasyczny.

<b>Algorytm klasyczny</b>	<b>ST zaimplementowane w systemie</b>
Bazuje na grafie, gdzie mamy węzły sell i buy	Bazuje na wywołaniach rekurencyjnych
Jest bardzo losowy – w wielu miejscach używa randoma	Jest deterministyczny
Umożliwia wymianę wielu elementów jednocześnie – skomplikowane, złożone zmiany	Umożliwia wymianę pojedynczych zleceń
W danym kroku agent losuje zlecenie które chciałby przyjąć	Agent próbuje przyjąć zlecenie wysłane przez poprzedniego agenta
Optymalizuje koszt globalnie	Optymalizuje koszt w kontekście zlecenia które napłynęło – dalszą optymalizację ma wykonać fullIST (w sensie wymiany zleceń pomiędzy jednostkami w celu globalnej optymalizacji – w danym czasie wymieniane jest pojedyncze zlecenie)
Szukanie dopasowania – złożone obliczeniowo	Złożonym obliczeniowo jest tworzenie nowych kalendarzy (sprawdzanie czy da się je utworzyć)
Zakłada miękkie okna czasowe (przynajmniej tak wynika z opisu)	Może działać z twardymi i miękkimi oknami czasowymi
Wymaga detekcji wadliwych rozwiązań	Zawsze tworzy poprawne rozwiązania
Ma możliwość rozproszenia	Scentralizowany

### 3.8.4. Wzorzec sytuacyjny

Jako że wyniki testów przeprowadzonych po implementacji algorytmów simulated trading dawały dość zróżnicowane wyniki, tzn. na ich podstawie nie dało się określić jednej optymalnej konfiguracji, dla wszystkich problemów, zaistniała konieczność użycia wzorca

sytuacyjnego. Jego wprowadzenie miało na celu automatyzację wyboru najlepszej konfiguracji, dla danego problemu.

W systemie istnieje możliwość wyboru trybu pracy. Możemy symulację uruchamiać jak poprzednio (podając wszystkie parametry), bądź użyć opcji „autoConfig=true”, co uruchamia mechanizm doboru konfiguracji na podstawie wzorców. Jeśli mechanizm ten jest włączony, to inne parametry podawane w konfiguracji, które dotyczą konfiguracji algorytmów, są ignorowane.

Sama implementacja wzorców znajduje się w pakiecie *pattern*. Mamy tam kilka klas:

- a) PatternCalculator – klasa służąca do obliczania wartości poszczególnych wzorców.

Obecnie możemy wyliczać następujące wartości:

- a. średnia z ilości ładunków w każdym zleceniu
- b. odchylenie standardowe z ilości ładunków w każdym zleceniu
- c. średnia z odległości między parami załadunku, wyładunku, oraz bazą
- d. odchylenie standardowe z odległości między parami załadunku, wyładunku, oraz bazą
- e. średnia długość okien czasowych
- f. średnia z najmniejszych odległości między poszczególnymi punktami załadunku/wyładunku (liczone dla każdego punktu)
- g. odchylenie standardowe z najmniejszych odległości między poszczególnymi punktami załadunku/wyładunku (liczone dla każdego punktu)
- h. odległość środka ciężkości zleceń od bazy
- i. średnia liczba zleceń mieszczących się w oknie czasowym każdego zlecenia
- j. max okno czasowe
- k. min okno czasowe
- l. średnia z większych okien czasowych (pickup lub delivery) w ramach każdego zlecenia
- m. średnia z mniejszych okien czasowych (pickup lub delivery) w ramach każdego zlecenia
- n. odchylenie standardowe z dystansów zleceń od bazy

- b) ConfigurationChooser – klasa, która jest używana do wyboru właściwej konfiguracji algorytmów
- c) ...Chooser – klasy, które implementują wybór właściwej konfiguracji dla każdego typu problemu

Obecnie wzorce zostały ustalone dla problemów pdp\_100, oraz pdp\_200.

Samo ich wyznaczanie przebiegało podobnie do budowania sieci neuronowych (z tą różnicą, że tutaj był to proces ręczny). Najpierw każdy problem (z grupy wzorcowej) został rozwiązyany przy pomocy algorytmów różnie skonfigurowanych. Potem dla każdego problemu wyliczone zostały wartości poszczególnych wzorców. W końcu wyznaczone zostały przedziały wartości odpowiednich wzorców (wykorzystany został jedynie wzorzec 4) na podstawie zestawienia wyników testów, jak również wartości wzorców.

Jeśli chodzi o konfiguracje które są wybierane automatycznie, to są to:

- d) brut1
  - algorytm – BruteForceAlgorithm
  - wybór “worstCommission” – waitTime
  - obliczanie kosztu zlecenia – po czasie dostarczenia zleceń
  - stopień zagłębienia ST – 8
- e) brut1\_dist
  - algorytm – BruteForceAlgorithm
  - wybór “worstCommission” – waitTime
  - obliczanie kosztu zlecenia – po przyroście dystansu (wykorzystanie metody getRatio)
  - stopień zagłębienia ST – 8
- f) brut2
  - algorytm – BruteForceAlgorithm2
  - wybór “worstCommission” – waitTime
  - obliczanie kosztu zlecenia – po czasie dostarczenia zleceń
  - stopień zagłębienia ST – 8

g) brut2\_dist

algorytm – BruteForceAlgorithm2

wybór “worstCommission” – waitTime

obliczanie kosztu zlecenia – po przyroście dystansu (wykorzystanie metody  
getRatio)

stopień zagębiania ST – 8

#### 1.8.4.1. Użycie poszczególnych miar do wyboru konfiguracji

Wybór konfiguracji następuje krokowo:

- a. Określenie rozmiaru problemu – ilość zleceń
- b. Określenie szerokości okien czasowych (przykładowo: pdp\_1XX czy pdp\_2XX) – wybór na podstawie iloczynu wskaźników 6 i 7, oraz na podstawie wskaźnika 14
- c. Określenie czy problem jest Ic, Ir, Irc – wykorzystanie wskaźnika 6
- d. Wybór właściwej konfiguracji – wskaźnik 4

### 1.9. Miary

System umożliwia obliczanie różnych miar w trakcie działania symulacji i zapisu ich do różnych formatów. Poniżej przedstawione zostały klasy odpowiadające za wyliczanie i zapis miar, oraz sposób ich konfiguracji.

#### 3.9.1. Wyliczanie miar

Do wyliczania poszczególnych miar służą klasy znajdujące się w pakiecie „measure”.

Każda z nich musi dziedziczyć po MeasureCalculator. Do wyliczania miar (metoda calculateMeasure) wykorzystujemy:

- oldSchedule – kalendarz przed uruchomieniem algorytmu ST (po dodaniu zlecenia)
- newSchedule – kalendarz po uruchomieniu algorytmu ST (może być null)
- info – zawiera między innymi informacje o bazie i deadlinie
- timestamp – aktualny timestamp (wartość jest aktualizowana przed każdym wywołaniem metody calculateMeasure)

Wszystkie kalkulatory miar po wczytaniu konfiguracji są przechowywane w MeasureCalculatorHolder. Ważne jest żeby w konfiguracji podawać pełną nazwę klasy

kalkulatora, oraz żeby znajdował się on w pakiecie measure (ponieważ do ich tworzenia używana jest refleksja).

Dzięki takiemu podejściu dodanie wyliczania nowej miary jest bardzo proste.

Wystarczy w pakiecie measure stworzyć nową klasę, która dziedziczy po MeasureCalculator – wtedy już możemy używać jej nazwy w pliku konfiguracyjnym i wszystko będzie działać.

### 3.9.2. Zapisywanie miar

Z uwagi na to, że zaistniała konieczność zapisywania wyliczanych miar w kilku formatach powstał pakiet „measure.printer”, który umożliwia dodawanie odpowiednich printerów do dowolnego formatu.

Każdy printer musi implementować interfejs MeasurePrinter. Wszystkie printery są przechowywane (po wczytaniu konfiguracji) w obiekcie klasy PrintersHolder.

Znaczenie poszczególnych metod interfejsu MeasurePrinter:

- createDocument – metoda wołana na samym początku (przed dostarczeniem danych o miarach)
- printColumns – ta metoda również jest wołana tylko raz. Nazwy kolumn są tworzone dynamicznie w zależności od skonfigurowanych miar. Zawsze pierwszym elementem jest holonId, a następnie nazwy poszczególnych miar (wartość zwracana przez metodę getName kalkulatora).
- printNextPart – przekazywana jest lista miar, które zostały wyliczone dla konkretnego zlecenia i timestamp'u – należy zauważyć, że poszczególne elementy listy odpowiadają kolejności miar, których nazwy zostały przekazane jako nazwy kolumn (poza holonId)
- finish

W tym przypadku bardzo ważne jest nazewnictwo. Każdy nowy printer musi nazywać się następująco <rozszerzenie drukowanymi literami>Printer i znajdować się w pakiecie measure.printer (no i oczywiście musi implementować interfejs MeasurePrinter). Elementy przed częścią nazwy Printer podajemy potem w konfiguracji jako formats (w pliku konf wielkość liter nie ma znaczenia).

Obecnie mamy zaimplementowane printery do formatów: xml i xls

### 3.9.3. Zaimplementowane kalkulatory miar

Poniżej znajduje się opis w pseudokodzie poszczególnych kalkulatorów.

Przypominam, że każdy z nich korzysta z dwóch map oldSchedules (z kalendarzami holonów, przed uruchomieniem ST), oraz newSchedules (po ST).

#### 3.9.3.1. Nowe miary

- a) **GivenCommissionsNumber** – ilość zleceń, które zostały oddane w wymianach

```
Measure result = new Measure();
//jeśli ST nie było uruchamiane, to żadne zlecenia nie były wymieniane
if(newSchedules == null)
    return result;

for każdy holon, którego kalendarz jest w mapie newSchedules
    policz ile jest zleceń, których nie ma w kalendarzu przed uruchomieniem ST, a
    które są po jego uruchomieniu. Otrzymaną wartość zapisz jako wartość miary
    dla
    holona

return result;
```

- b) **AverageDistPerCommissionBeforeChange** – średni dystans potrzebny na zrealizowanie jednego zlecenia z kalendarza. Wartość jest liczona następująco – obliczana jest suma dystansów z bieżącego położenia jednostki przez wszystkie zlecenia i z powrotem do bazy i otrzymana wartość jest dzielona przez ilość zleceń w kalendarzu. Wartość jest wyliczana dla kalendarza przed uruchomieniem algorytmu ST

```
Measure result = new Measure();

if(oldSchedules==null)
    return result;

currentLocation = bieżące położenie holona
dist = 0
for każdy holon (h) z mapy oldSchedules
    for każde zlecenie (com) z kalendarza bierzącego holona h
        nextLocation = lokacja zlecenia com
        dist += odległość między currentLocation a nextLocation
        currentLocation = nextLocation
    dist += odległość między currentLocation a bazą
    dodaj wartość miary jako dist/ilość zleceń w kalendarzu holona h
```

- c) **AverageDistPerCommissionAfterChange** – jak wyżej, tylko wyliczane po uruchomieniu algorytmu ST (w pseudokodzie oldSchedules jest zastąpione przez newSchedules)
- d) **NumberOfCommissionsWeCanAddToOthersBeforeChanges** – ilość zleceń, które moglibyśmy wstawić do kalendarza innego holonu (obliczane przed uruchomieniem ST)

```

Measure measure = new Measure()
if(oldSchedules == null)
    return measure;
for każdy kalendarz (s) z mapy oldSchedules
    value=0
    for każde zlecenie (com) z kalendarza s
        for każdy kalendarz (s2) z mapy oldSchedules
            if s==s2
                continue
            if można wstawić com do kalendarza s2
                value++
                break
    dodaj wartość value jako wartość miary dla bieżącego holona

```

- e) **NumberOfCommissionsWeCanAddToOthersAfterChanges** – jak wyżej tylko obliczane po uruchomieniu ST (w pseudokodzie mapa oldSchedules będzie zastąpiona przez newSchedules)
- f) **NumberOfCommissionsOthersCanAddToUsBeforeChanges** – ilość zleceń, które moglibyśmy wstawić do własnego kalendarza, od innych holonów (obliczane przed uruchomieniem alg. ST)

```

Measure measure = new Measure()
if(oldSchedules == null)
    return measure;
for każdy kalendarz (s) z mapy oldSchedules
    value=0
    for każdy kalendarz (s2) z mapy oldSchedules
        if s==s2
            continue
        for każde zlecenie (com) z kalendarza s2
            if można wstawić com do kalendarza s
                value++
                break
    dodaj wartość value jako wartość miary dla bieżącego holona

```

g) **NumberOfCommissionsOthersCanAddToUsAfterChanges** – jak wyżej tylko po ST

### 3.9.3.2. Miary używane wcześniej do wzorców dla holonów

Miary wymienione poniżej są wyliczane w dwóchwersjach. Raz dla wszystkich zleceń z kalendarza (zrealizowanych i nie), a raz z uwzględnieniem tylko niezrealizowanych jeszcze zleceń. Nazewnictwo jest takie: AllCommissions – gdy brane pod uwagę są wszystkie zlecenia (zrealizowane i nie), oraz UndeliveredCommissions (gdy brane pod uwagę są tylko zlecenia jeszcze niezrealizowane).

W pseudokodach pokazane jest tylko, w jaki sposób tworzona jest lista wartości (double) do wyliczenia średniej, lub odchylenia standardowego. Znaczenie parametrów:

- commissionsFromSchedule – zlecenia wzięte z kalendarza holona (w zależności czy liczymy dla wszystkich zleceń, czy tylko dla jeszcze niezrealizowanych, przyjmuje różne wartości).
- commissions – zlecenia, które przyszły od dystrybutora i które jeszcze nie zostały nikomu przydzielone

a) **AverageLoadFromAllCommissions / AverageLoadFromUndeliveredCommissions**

– średnia ilość ładunków w każdym zleceniu

```
For każde zlecenie (com) z listy commissionsFromSchedule  
    values.add(com.getLoad())  
for każde zlecenie (com) z listy commissions  
    values.add(com.getLoad())
```

#### Uwaga!

Dalej założone jest, że lista coms zawiera zlecenia z list: commissionsFromSchedule, oraz commissions. Ponadto pseudokody, będą dotyczyć wartości dla pojedynczego holona

b) **AverageDistanceFromCurLocationToBaseForAllCommissios /**

**AverageDistanceFromCurLocationToBaseForUndeliveredCommissions** – średnia z odległości między bieżącym położeniem holonu, parami załadunku i wyładunku oraz bazą

```
for każde zlecenie (com) z listy coms
    dist = 0
    dist += odległość między bieżącym położeniem holona, a składową pickup
    zlecenia
        com
        dist += odległość między lokacjami dla pickup i delivery zlecenia com
        dist += odległość między klawową delivery zlecenia com, a bazą
        values.add(dist)
return values
```

- c) **StandardDeviationOfDistanceFromCurLocationToBaseForAllCommissions / StandardDeviationOfDistanceFromCurLocationToBaseForUndeliveredCommissions** – odchylenie standardowe z odległości między bieżącym położeniem holonu, parami załadunku i wyładunku oraz bazą. Pseudokod taki jak wyżej.

- d) **AverageTimeWindowsSizeForAllCommissions / AverageTimeWindowsSizeForUndeliveredCommissions** – średnia długość okien czasowych

```
for każde zlecenie (com) z listy coms
    values.add(długość okna dla com-pickup)
    values.add(długość okna dla com-delivery)
return values
```

- e) **AverageMinDistBetweenAllCommissions / AverageMinDistBetweenUndeliveredCommissions** – średnia z najmniejszych odległości między poszczególnymi punktami załadunku / wyładunku (liczone dla każdego punktu)

```
For każde zlecenie (com) z listy coms
    location = com-pickup
    values.add(odległość między location, a getNearestLocation(location, com-
    pickup-id))
    location = com-delivery
    values.add(odległość między location, a getNearestLocation(location, com-
    delivery-
                                id))
return values
```

```
getNearestLocation(location, id)
```

```

for każde zlecenie (com) z listy coms
    dist = odległość między location, a com-pickup
    if dist < bestDist
        bestDist = dist
        nearestLocation = com-pickup
    dist = odległość między location, a com-delivery
    if dist < bestDist
        bestDist = dist
        nearestLocation = com-delivery
return nearestLocation

```

- f) **StandardDeviationOfMinDistBetweenAllCommissions / StandardDeviationOfMinDistBetweenUndeliveredCommissions** – odchylenie standardowe z najmniejszych odległości między poszczególnymi punktami załadunku / wyładunku (liczone dla każdego punktu). Pseudokod jak wyżej.

- g) **DistanceFromCenterOfGravityOfAllCommissionsToHolon / DistanceFromCenterOfGravityOfUndeliveredCommissionsToHolon** – odległość środka ciężkości zleceń od holona

```

for każde zlecenie (com) z listy coms
    x.add(com-pickupX)
    x.add(com-deliveryX)
    y.add(com-pickupY)
    y.add(com-deliveryY)
return dystans między bieżącym położeniem holona a punktem o współrzędnych
(avg(x), avg(y))

```

- h) **AverageNumberOfComsWithinTimeWinOfAllCommissions / AverageNumberOfComsWithinTimeWinOfUndeliveredCommissions** – średnia liczba zleceń mieszczących się w oknie czasowym każdego zlecenia.

```

for każde zlecenie (com) z listy coms
    values.add(getCommissionsBetweenTimeWindow(coms, com.pickupTime1,
                                                com.pickupTime2)
    values.add(getCommissionsBetweenTimeWindow(coms, com.deliveryTime1,
                                                com.deliveryTime2)
return average(values)

getCommissionsBetweenTimeWindow(com,...)
    for każde zlecenie (com2) z listy coms
        if okno czasowe zlecenia com2-pickup mieści się w oknie zlecenia com

```

```

        value++
        if okno czasowe zlecenia com2-delivery mieści się w oknie zlecenia
com
        value++
return value
    
```

i) **AverageMaxTimeWinSizeForAllCommissions /**

**AverageMaxTimeWinSizeForUndeliveredCommissions** – średnia z czasu od rozpoczęcia okna czasowego załadunku do zakończenia okna czasowego wyładunku.

```

for każde zlecenie (com) z listy coms
    values.add(com-deliveryTime2 – com-pickupTime1_)
    
```

j) **AverageMinTimeWinSizeForAllCommissions /**

**AverageMinTimeWinSizeForUndeliveredCommissions** – średnia z czasu od zakończenia okna czasowego załadunku do rozpoczęcia okna czasowego wyładunku.

```

for każde zlecenie (com) z listy coms
    values.add(com-deliveryTime1 – com-pickupTime2)
    
```

k) **StandardDeviationAllComsFromHolon /**

**StandardDeviationUndeliveredComsFromHolon** – odchylenie standardowe z dystansów zleceń od bieżącej pozycji holona.

```

for każde zlecenie (com) z listy coms
    values.add(dystans między bieżącą lokacją holona, a com-pickup)
    values.add(dystans między bieżącą lokacją holona, a com-delivery)
    
```

### 3.9.3.3. Miary związane z miękkimi oknami

Tutaj, tak jak poprzednio przedstawiono pseudokody dotyczące pojedynczego holonu.

a) **SummaryLatency** – całkowite spóźnienie.

```

time = creationTime
currentLocation = depot
for każde zlecenie (com) z kalendarza
    nextLocation = położenie zlecenia com
    
```

```

dist = dystans między current i next location
if time + dist > koniec okna czasowego zlecenia com
    latency += time + dist - koniec okna czasowego zlecenia
com
if time + dist < początek okna czasowego com
    time = początek okna czasowego com
else
    time += dist
currentLocation = nextLocation
return latency

```

**b) AverageLatencyPerCommission** – średnie spóźnienie na zlecenie.

```

time = creationTime
currentLocation = depot
summaryLatency = 0.0
for każde zlecenie (com) z kalendarza
    nextLocation = położenie zlecenia com
    dist = dystans między current i next location
    if time + dist > koniec okna czasowego zlecenia com
        latency = time + dist - koniec okna czasowego zlecenia com
        summaryLatency += latency
    if time + dist < początek okna czasowego com
        time = początek okna czasowego com
    else
        time += dist
    currentLocation = nextLocation
return summaryLatency/ilość zleceń

```

**c) MaxLatency** – maksymalne spóźnienie.

```

time = creationTime
currentLocation = depot
maxLatency = 0.0
for każde zlecenie (com) z kalendarza
    nextLocation = położenie zlecenia com
    dist = dystans między current i next location
    if time + dist > koniec okna czasowego zlecenia com
        latency = time + dist - koniec okna czasowego zlecenia com
        maxLatency = max(maxLatency, latency)
    if time + dist < początek okna czasowego com
        time = początek okna czasowego com
    else
        time += dist
    currentLocation = nextLocation
return maxLatency

```

d) **PercentageOfDelayComs** – procent spóźnionych zleceń.

```

time = creationTime
currentLocation = depot
delayComs = 0
for każde zlecenie (com) z kalendarza
    nextLocation = położenie zlecenia com
    dist = dystans między current i next location
    if time + dist > koniec okna czasowego zlecenia com
        delayComs++
    if time + dist < początek okna czasowego com
        time = początek okna czasowego com
    else
        time += dist
    currentLocation = nextLocation
return delayComs/ilość zleceń * 100

```

e) **WaitTime** – sumaryczny WaitTime.

```

time = creationTime
currentLocation = depot
waitTime = 0.0
for każde zlecenie (com) z kalendarza
    nextLocation = położenie zlecenia com
    dist = dystans między current i next location
    if time + dist < początek okna czasowego com
        waitTime += początek okna czasowego com - (time + dist)
        time = początek okna czasowego com
    else
        time += dist
    currentLocation = nextLocation
return waitTime

```

f) **MaxWaitTime** – maksymalny WaitTime.

```

time = creationTime
currentLocation = depot
maxWaitTime = 0.0
for każde zlecenie (com) z kalendarza
    nextLocation = położenie zlecenia com
    dist = dystans między current i next location
    if time + dist < początek okna czasowego com
        waitTime = początek okna czasowego com - (time + dist)
        maxWaitTime = max(maxWaitTime, waitTime)
        time = początek okna czasowego com

```

```
else
    time += dist
    currentLocation = nextLocation
return maxWaitTime
```

### 3.9.4. Wizualizacja miar

System posiada dwie opcje wizualizacji miar:

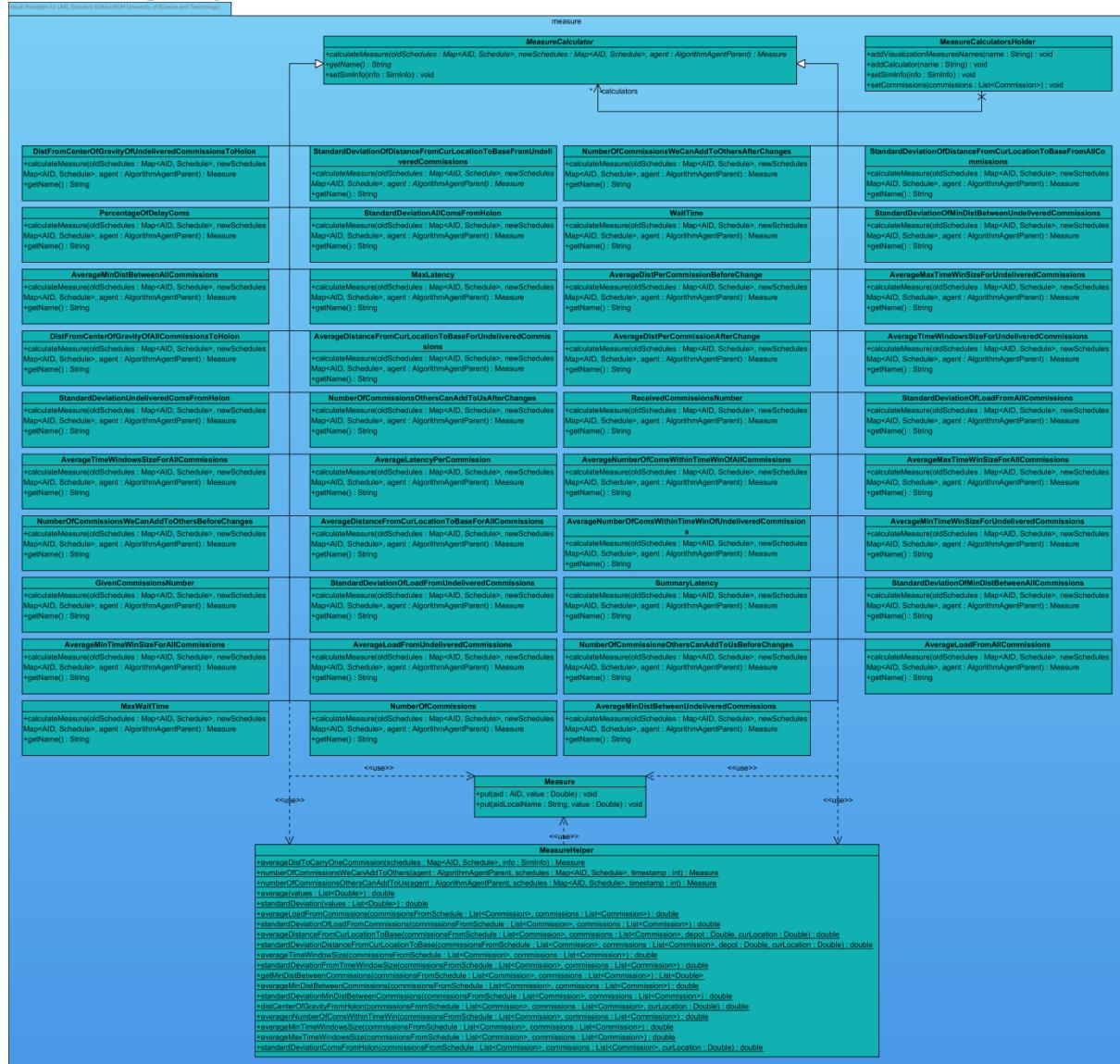
- w trakcie działania
- na podstawie pliku z zapisanymi miarami

Żeby można było wizualizować daną miarę w czasie działania systemu, wystarczy ustawić atrybut visualize elementu <measure> na true.

Jeśli chcemy uruchomić wizualizację miar z pliku, to najpierw musimy je wygenerować (FORMATEM MUSI BYĆ „xml”). Następnie wystarczy wybrać wygenerowany plik w oknie, które pojawi się po uruchomieniu measureVisualization.bat

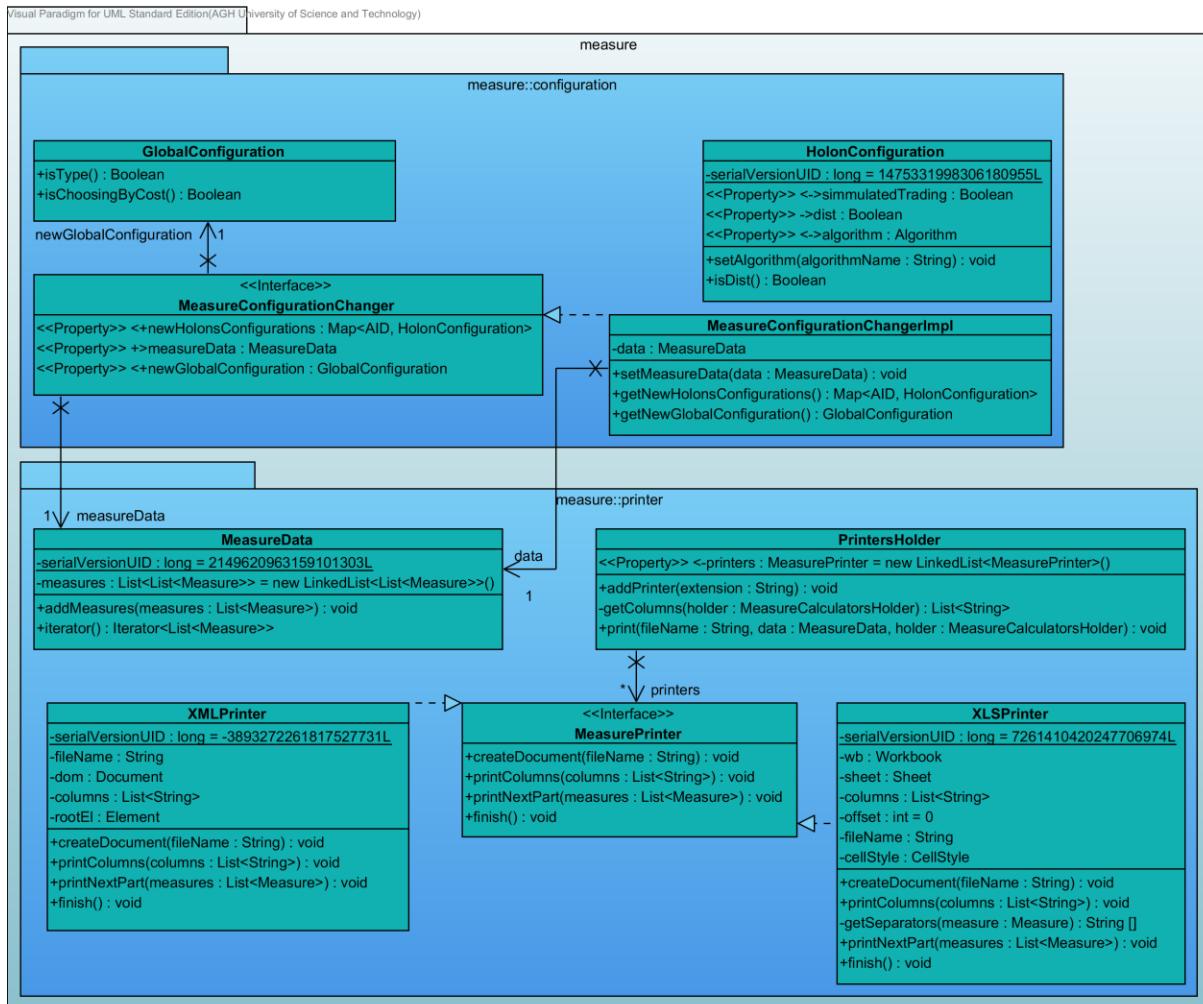
# Dispatch Rider – dokumentacja projektowa

## 3.9.5. Diagramy

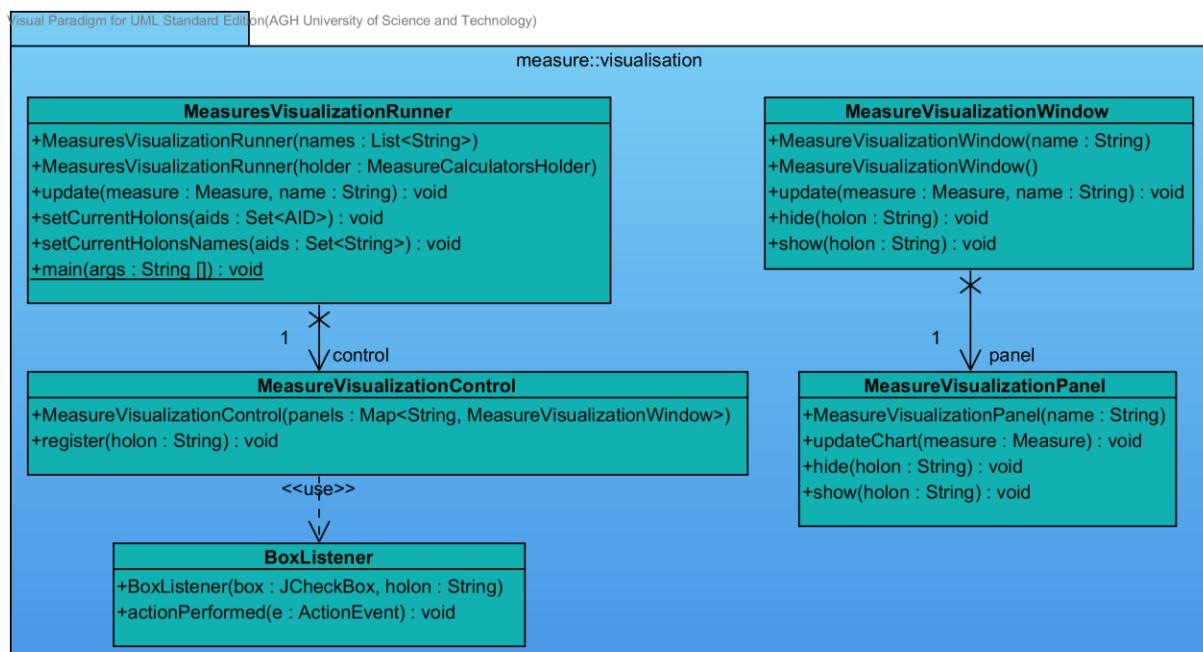


Rysunek 26 Diagram klas przedstawiający mechanizm zarządzania miarami

## Dispatch Rider – dokumentacja projektowa



Rysunek 27 Diagram klas przedstawiający mechanizm zarządzania miarami 2



Rysunek 28 Diagram klas przedstawiający mechanizm zarządzania miarami 3

### 3.10. Dynamiczna zmiana konfiguracji

System umożliwia dynamiczną zmianę konfiguracji w czasie działania symulacji na podstawie wyliczanych miar. Wszystkie niezbędne do tego klasy znajdują się w pakiecie „*measure.configuration*”.

Do zmiany konfiguracji wykorzystuje się klasy, które implementują interfejs *MeasureConfigurationChanger*. Obecnie system został skonfigurowany w taki sposób, że wykorzystuje implementację tego interfejsu *MeasureConfigurationChangerImpl*.

Konfiguracja, którą możemy ustawać, dzieli się na dwie części:

- *GlobalConfiguration* – konfiguracja, która jest ustawiana dla wszystkich holonów (całej symulacji). Zmiany dotyczą tylko i wyłącznie dystrybutora i sposobu jego działania. Możemy modyfikować:
  - Sposób wysyłania zleceń (paczkami, lub po kolej)
  - Sposób przydziału zleceń (z preferowanym kosztem, lub wypełnieniem istniejących holonów)
  - Ilość uruchomień algorytmów ST
  - Stopień zagłębiania w algorytmie complexSimmulatedTrading
  - W jaki sposób ma być wybierane zlecenie do wymiany (worstCommission)  
– biorąc pod uwagę dystans, czy czas
- *HolonConfiguration* – konfiguracja ustawiana dla konkretnego holonu (każdy może mieć inne parametry). Tutaj możemy modyfikować:
  - Algorytm rozdziału zleceń
  - Sposób wyliczania kosztu nowego zlecenia
  - Włączanie/wyłączanie simmulated trading w ramach jednostki

Powyższe parametry odpowiadają tym używanym w konfiguracji symulacji, więc nie będę ich ponownie opisywać.

Należy tutaj zaznaczyć, że nie musimy ustawiać wszystkich parametrów we wspomnianych wyżej klasach. Nieustawione parametry zostaną pominięte.

### Konfiguracja:

Żeby uruchomić zmianę konfiguracji musimy w głównym pliku konfiguracyjnym dla elementu *commissions* ustawić wartość atrybutu *confChange* na prawdę.

## 3.11. Miękkie okna czasowe

Do tej pory każde zlecenia (zarówno pickup jak i delivery) posiadało ścisłe określone dwa czasy:

- Najwcześniejszy czas rozpoczęcia realizacji zlecenia – jeśli jednostka przyjechała przed tym czasem, to musiała czekać, aż czas ten zostanie osiągnięty
- Najpóźniejszy czas realizacji zlecenia – określał czas, do kiedy musimy dojechać do wyznaczonej lokalizacji

Nas szczególnie interesuje najpóźniejszy czas realizacji zlecenia. Czas ten określał, czy zlecenia mogły zostać zrealizowane przez daną jednostkę (czy zdąży dojechać na czas). Takie podejście jest nazywane twardymi oknami czasowymi. Nas interesuje nieco inna sytuacja. Chcemy umożliwić pojazdom możliwość spóźnienia się (przyjazdu po najpóźniejszym czasie realizacji zlecenia). Oczywiście takie spóźnienie (tak jak w realnym świecie) wiąże się z kosztem. W tym celu została wprowadzona funkcja kary, o której później.

Wprowadzenie miękkich okien czasowych wiązało się z modyfikacją wszystkich algorytmów przydziału zleceń, jak i algorytmów ST. Kolejne rozdziały zawierają opisy dotyczące działania systemu w trybie z miękkimi oknami czasowymi, oraz sposób jego konfiguracji.

### 3.11.1. Funkcja kary

Klasa odpowiadająca za ewaluację funkcji kary to punishment.PunishmentFunction. Funkcja kary jest funkcją złożoną, co oznacza, że możemy ją definiować za pomocą kilku przepisów. Jedynym ograniczeniem jest to, że wartości funkcji muszą być dodatnie, oraz musi ona być określona na zbiorze liczb rzeczywistych dodatnich ( $x>0$ ).

- Sam wzór podajemy w konfiguracji. Jego budowa jest następująca:

```
<przepis_1>?<przepis_2>?...?<przepis_n> - czyli konkretne przepisy oddzielamy znakiem '?'
```

- Każdy przepis definiujemy, jako:

```
<wzór>;<warunek>
```

- <wzór> - dowolny wzór funkcji, który jako zmienną używa 'latency' np. 2\*latency (latency – opóźnienie)
- <warunek> - dowolny warunek logiczny (można używać wszystkich operatorów java - &&, ||, !=, ==, (, ) ). Oczywiście w warunku musimy używać zmiennych używanych we wzorze np. latency >=8
- Jeśli nie podamy warunku (pominiemy też ';'), to wzór jest określony dla wszystkich liczb rzeczywistych dodatnich (warunek ma postać 'true')
- Przykładowa definicja funkcji kary:

```
latency;latency<=10?latency*latency;latency>10
```

### 3.11.2. Wartości domyślne – wagi zleceń

W funkcji kary możemy używać dowolnych zmiennych np: ala, ola. Muszą one jednak potem mieć zdefiniowane wartości domyślne w konfiguracji. Wartości tych parametrów mogą różnić się dla konkretnych zleceń (można je widzieć jako wagi zleceń). Wagi są definiowane w pliku z def zleceń. Format zapisu wartości domyślnych, jak i wartości wag w pliku z def zleceń jest taki sam tzn:

```
<identyfikator_1>=<wartość_1>;<identyfikator_2>=<wartość_2>;...
```

Wartości wag w pliku ze zleceniami wprowadzamy po znaku tabulacji po ostatnim parametrze. W pliku tym nie musimy podawać wszystkich wag (możemy nie podawać ich wcale), wtedy dla danego zlecenia zostaną użyte wartości domyślne.

W przypadku błędów, np. w nazwie identyfikatora, lub nie podaniu wartości domyślnej dla jakiegoś identyfikatora błąd pojawi się w trakcie symulacji podczas próby wyliczenia wartości funkcji kary.

### 3.11.3. Zmiany w implementacji algorytmów przydziału zleceń i ST

Zmiany polegały na tym, żeby umożliwić wprowadzenie miękkich okien czasowych.

Wiązało się to z koniecznością wprowadzenia dodatkowej metody, która wylicza wartość opóźnienia (jeśli używamy trybu z miękkimi oknami) i potem uwzględniania tej wartości podczas wyliczania kosztu realizacji zlecenia. Należało również zmodyfikować sposób określania czy warunki symulacji nie zostały naruszona (czy nie przekroczone najpóźniejszego czasu przyjazdu dla twardych okien) – tutaj prace polegały na tym, żeby wersja dla twardych i miękkich okien nie różniła się zbytnio.

Priorytetem było to, żeby umożliwić działania systemu, bez miękkich okien czasowych (tak jak poprzednio). Udało się to zrealizować. Podczas symulacji (na starej konfiguracji) system działa jak poprzednio (nawet trochę udało się go przyspieszyć). Po włączeniu miękkich okien czasowych system bardzo zwalnia. Jest to zrozumiałe zachowanie, ponieważ przeglądamy dużo więcej rozwiązań (bo nie odrzucamy ich już tak szybko jak poprzednio, gdyż naruszenie ograniczeń czasowych nie eliminuje zleceń).

We wszystkich algorytmach pozostawione zostało ninaruszalne kryterium powrotu do bazy przed upływem określonego czasu (deadline).

Funkcja kary jest używana w następujących miejscach w systemie:

- algorytmy rozdziału zleceń – korzystają bezpośrednio z metody Schedule calculateCost, która uwzględnia funkcję kary
- algorytmy ST – jak wyżej
- algorithm.Schedule – tutaj najwięcej jest używana funkcja kary. Podczas jej wprowadzania starałem się, żeby wszystkie jej użycia były właśnie z tej klasy. Jest to spowodowane tym, że do wyliczenia kary musimy mieć spóźnienie, które może być wyliczone na podstawie danych znajdujących się w tej klasie.

### 3.11.4. Sposób działania

W konfiguracji miękkich okien musimy podać parametr (holons) określający max liczbę jednostek, które jesteśmy w stanie zaakceptować jako dobre rozwiązanie. Znaczenie tego parametru w sensie działania systemu jest następujące. Do momentu, gdy ilość holonów jest mniejsza, niż holons, system działa w trybie przydzielania zleceń po koszcie [7.1], natomiast potem przełącza się na tryb z preferowanym załadunkiem [7.2]. Takie podejście miało zapobiec sytuacji, gdy pierwszy stworzony holon ma przydzielane wszystkie

zlecenia, dopóki nie narusza deadline'u. W tym przypadku mógłby on mieć tak niekorzystny kalendarz, że nie dałoby się go już naprawić w wymianach ST.

### 3.12. Optymalizacja algorytmu complexSimulatedTrading

Do tej pory algorytm complex ST działał następująco: jeśli udało się znaleźć taką konfigurację zleceń, że nowe zlecenie mogło zostać zrealizowane, przez aktualnie istniejące holony, to algorytm kończył działanie na pierwszej znalezionej konfiguracji.

Optymalizacja algorytmu polegała na tym, aby umożliwić działanie tego algorytmu w dwóch trybach. Pierwszy, który był używany dotychczas, oraz drugim w którym wybierana byłaby optymalna (pod względem kosztu) konfiguracja zleceń.

Żeby to osiągnąć wprowadzony został dodatkowy parametr określający bieżące rozwiązanie. W przypadku znalezienia dobrej konfiguracji zleceń algorytm jest kontynuowany, a znalezione rozwiązanie jest porównywane z najlepszym dotychczas znalezionym.

#### Konfiguracja:

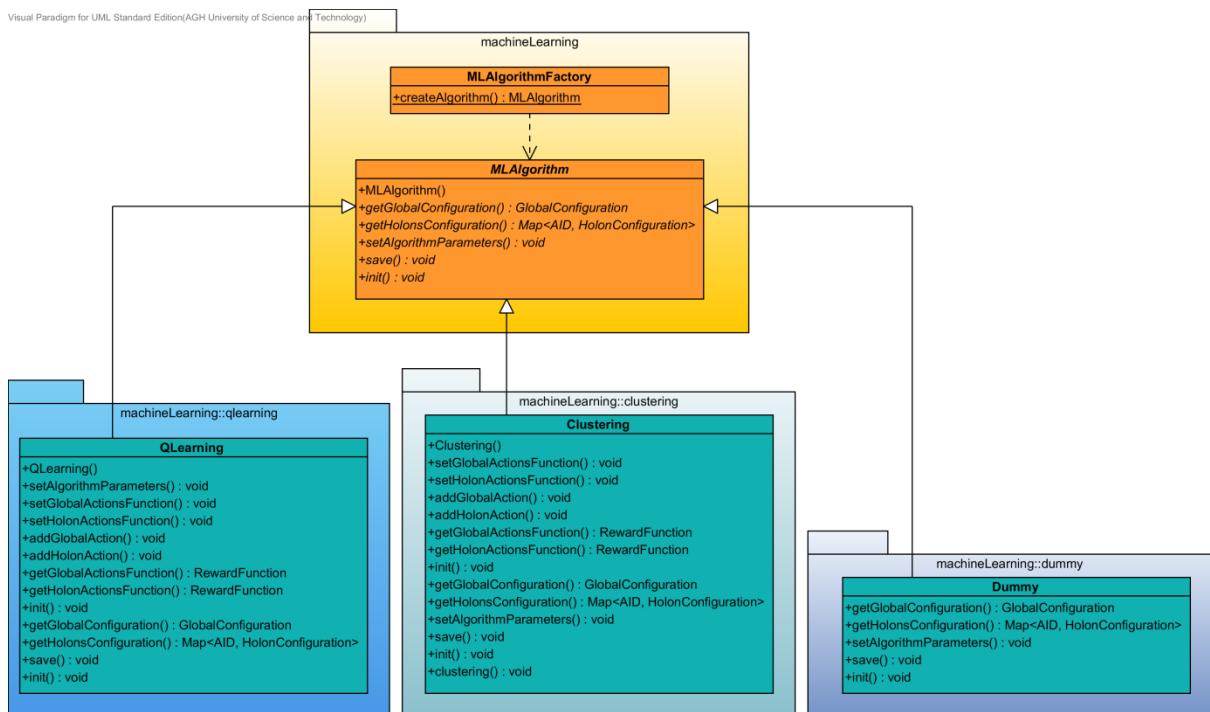
Sposób działania algorytmu jest konfigurowalny poprzez ustawienie atrybutu *firstComplexSTResultOnly*. Jeśli jego wartość jest ustawiona na true (wartość domyślna), to algorytm działa tak jak do tej pory, czyli wybierana jest pierwsza znaleziona konfiguracja, w przeciwnym wypadku poszukiwana jest optymalna konfiguracja.

Można by się było zastanowić, po co nam dwa tryby działania. Niestety wersja optymalna jest również bardziej czasochłonna, dlatego zapadła decyzja o pozostawieniu poprzedniego trybu działania.

### 3.13. Uczenie maszynowe

Z uwagi na to, że nasz system pozwala na wyliczanie różnych miar podczas symulacji, oraz na dynamiczną zmianę konfiguracji systemu w trakcie działania pojawił się pomysł na wprowadzenie uczenia maszynowego do systemu. Sama idea była bardzo prosta. Chcielibyśmy, żeby system sam (w trakcie działania) przełączał się na optymalną w danej

chwili konfigurację, co mogłoby skutkować zmniejszeniem czasu obliczeń, oraz polepszeniu otrzymywanych rozwiązań.



Rysunek 29 Diagram klas przedstawiający mechanizm uczenia maszynowego

### 3.13.1. QLearning

Z uwagi na wymienione we wstępie cechy systemu został wprowadzony „Q Learning”. Kolejne rozdziałły zawierają jego opis w odniesieniu do realizacji w systemie.

*Q Learning* polega na sekwencyjnym uczeniu systemu na podstawie odpowiednio przygotowanej tabeli, której wierszami są stany systemu, a kolumnami akcje, które mogą być wykonywane w danym stanie. Wydzielenie odpowiednich stanów i akcji jest jednym z najtrudniejszych zadań, gdyż wymaga dogłębnego zrozumienia i poznania dziedziny problemu. Ponadto nigdy nie wiemy, czy nie istnieje jakiś lepszy dobór stanów i akcji.

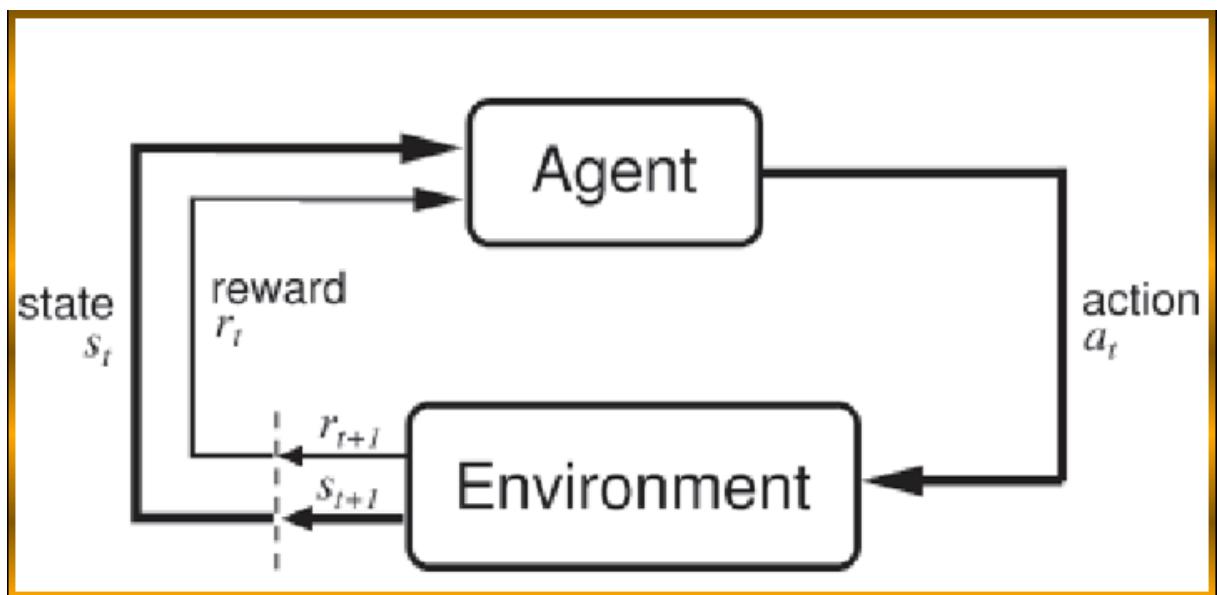
Z uwagi na to, że w systemie możemy zmieniać konfigurację globalnie, lub w ramach pojedynczych jednostek, mamy dwie tabelki. Pierwsza opisuje zmiany konfiguracji globalnej, a druga na poziomie holonu. W kolejnych podrozdziałach będą pokazane różnice między tymi tabelami.

Mimo wszystko ogólna zasada działania pozostaje taka sama i wygląda następująco:

- Na początku symulacji wczytywany jest plik z opisem tabel

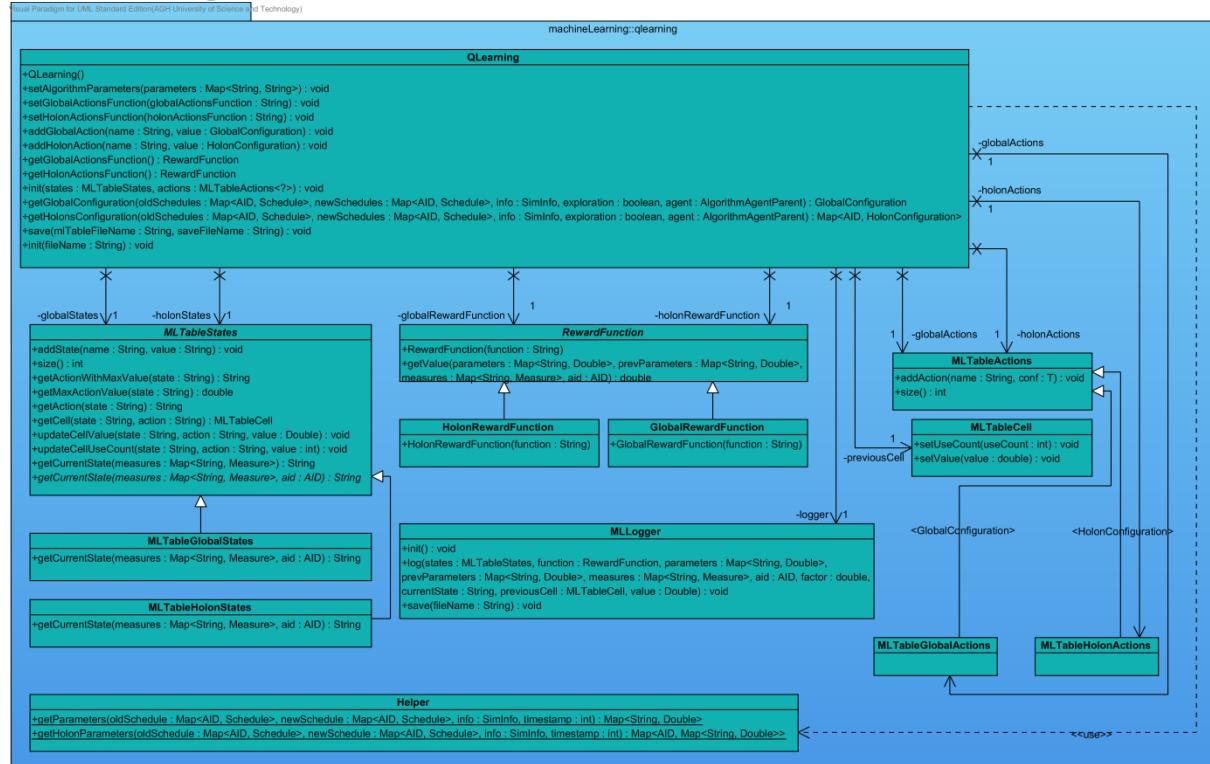
- dokonywana jest inicjalizacja tabeli (na podstawie zawartości poprzednio wczytanego pliku)
- po przybyciu każdego nowego zlecenia:
  - wyliczane są wartości miar i wybierany jest aktualny stan systemu ( $s'$ )
  - wybierana jest akcja dla aktualnego stanu ( $a'$ )
  - wyliczana jest nagroda (reward) po zmianie stanu  $s$  na  $s'$
  - wartość komórki dla  $(s, a)$  jest updateowana przez nową wartość, która zależy od wybranego trybu pracy [19.3.2]
  - $s = s'$ ,  $a = a'$
- po zakończeniu symulacji zawartość tabeli jest zapisywana do pliku (żeby można było użyć jej w kolejnej symulacji)

Poniższy rysunek bardzo dobrze obrazuje mechanizm QLearningu



Rysunek 30 Wizualizacja działania QLearningu

### 3.13.1.1. Diagram



Rysunek 31 Diagram klas przedstawiający mechanizm QLearningu

### 3.13.2. Stany

Teoretycznie nie mamy ograniczeń co do definicji stanu systemu. Może to być wszystko, co da się w jakiś sposób wyodrębnić. My przyjęliśmy jednak, że definiowanie stanów będzie opierać się na wartościach miar, wyliczanych w trakcie symulacji.

#### 3.13.2.1. Definiowanie stanów

Definicja stanów nieco różni się w tabeli z konfiguracją globalną i tą dotyczącą pojedynczego holonu. Ogólnie chodzi tutaj o to, że podczas definicji stanu dla konfiguracji globalnej musimy agregować wartości miar, natomiast przy definicji tabelki dla holonów możemy używać dodatkowo wartości miary dla konkretnej jednostki.

Kolejne stany definiujemy jako przedziały wartości. Możemy tutaj używać dowolnych operatorów logicznych, jak również matematycznych. Dzięki temu każdy stan może być zdefiniowany przez kilka różnych miar (a także przez ich kombinacje – np.: sumę).

Przykładowa definicja stanu:

avg(MaxWaitTime) > 100

Powyższy zapis należy rozumieć następująco:

System znajduje się w zdef stanie, jeśli wartość miary o nazwie MaxWaitTime zagregowana przy użyciu agregatora avg będzie większa od 100. Należy pamiętać, że przy definicji xml'owej powyższa def wygląda następująco: avg(MaxWaitTime) &gt; 100

Krótko mówiąc, podczas definiowania stanów możemy używać dowolnych miar z pakietu *measures* oraz agregatorów z pakietu *machineLearning.aggregator*.

### 3.13.2.2. Agregatory

Do zdefiniowania stanów niezbędne jest użycie odpowiedniego agregatora. Wszystkie agregatory znajdują się w pakiecie *machineLearning.aggregator*. W celu stworzenia nowego agregatora wystarczy w ww. pakiecie stworzyć klasę rozszerzającą klasę abstrakcyjną *machineLearning.aggregator.MLAgregator*. Musimy zaimplementować dwie metody:

- aggregate – jako parametr przyjmuje nazwę miary i zwraca zagregowaną wartość.  
W tej metodzie powinniśmy skorzystać z map: measures (mapa z wyliczonymi miarami, której kluczami są nazwy miar), oraz values, która służy jako cache (względy optymalizacyjne – gdyż daną wartość możemy chcieć użyć wiele razy)
- getName – zwraca nazwę agregatora. Nazwa jest bardzo ważna, gdyż jest potem używana podczas wstawiania odpowiednich wartości w def stanu.

Po stworzeniu nowego agregatora możemy go już używać w definicjach stanów.

### 3.13.2.3. Definiowanie akcji

Akcjom w naszym systemie odpowiadają zmiany konfiguracji. Z uwagi na to, że w konfiguracji globalnej mamy inne parametry, niż w konfiguracji dla holonów definicje dla każdej z tabel się znacząco różnią.

Jeśli chodzi o konfigurację globalną to możemy ustawać:

- sendingType – true/false – określa czy wysyłamy zlecenia jedno po drugim (false), czy paczkami (true)
- choosingByCost – true/false – określa tryb przydzielania zleceń (po koszcie, czy z preferencją wypełnienia istniejących jednostek)

- simmulatedTrading – int – ilość uruchomień simmulatedTrading
- STDepth – int – stopień zagłębienia w algorytmie complexST
- chooseWorstCommission –  
time/wTime/timeWithPunishment/distWithPunishment – sposób wyboru najgorszego zlecenia dla simulated trading

Jeśli chodzi o konfigurację holonów to możemy ustawić:

- algorithm – BruteForceAlgorithm/BruteForceAlgorithm2 – rodzaj algorytmu rozdziału zleceń
- newCommissionCostByDist – true/false – określa czy do wyznaczenia kosztu realizacji zlecenia przez holon ma być brany pod uwagę dystans, czy czas

Zmianie podlegają tylko ustawione parametry!

#### **3.13.2.4. Wybór akcji**

Jeżeli używamy trybu bez eksploracji, to wybierana jest akcja, z największą wartością komórki tabeli w danym stanie, w którym znajduje się system.

W przypadku używania trybu eksploracji akcja dla danego stanu jest wybierana następująco:

- dla każdej komórki w stanie, w którym znajduje się system wyliczane jest prawdopodobieństwo wg wzoru:

$$P(a_i | s) = \frac{k^{\bar{Q}(s,a_i)}}{\sum_j k^{\bar{Q}(s,a_j)}}$$

gdzie:

k – współczynnik ustawiany w definicji tabeli

Q – wartość komórki

- zgodnie z prawdopodobieństwami wybierana jest akcja

### 3.13.2.5. Używane stany i akcje w konfiguracji globalnej

Do pierwszych testów z machine learningu użyto następujących stanów i akcji:

#### Stany:

- Czynnik opisujący rozproszenie: małe/średnie/duże -  
 $\text{avg}(\text{AverageMinDistBetweenAllCommissions})$  o wartościach z przedziałów: <0, 2.63)/<2.63, 7.89)/<7.89,...)
- Czynnik opisujący luzy czasowe w trasach: małe/duże -  $\text{avg}(\text{WaitTime})$  o wartościach z przedziałów: <0, 10)/<10,...)
- Czynnik opisujący nakładanie się okien czasowych: małe/duże -  
 $\text{avg}(\text{AverageNumberOfComsWithinTimeWinOfAllCommissions})$  o wartościach z przedziałów: <0,4)/<4,...)

Kombinacje powyższych stanów dały łącznie 12 stanów

#### Akcje:

- Zmiana STDepth na: 0, 3, 8
- Zmiana sposobu przydzielania zleceń: preferowany koszt, preferowane wypełnienie
- Wł/wył simmulated trading

Kombinacje powyższych dały 12 akcji

#### Funkcja nagrody:

$2 * \text{holonsCount}/\text{holonsCount} + \text{bestDist}/\text{dist} + \text{_costOfCommission}/\text{costOfCommission}$

n

### 3.13.2.6. Używane stany i akcje w konfiguracji holonów

#### Stany:

- stany zostały zdefiniowane identycznie jak przy konfiguracji globalnej, z tą różnicą że opuszczone zostały aggregatory. Amieniona została tylko miara

AverageMinDistBetweenAllCommissions na  
AverageMinDistBetweenUndeliveredCommissions

**Akcje:**

- włączenie/wyłączenie simulated trading lokalnie w holonie
- zmiana algorytmu rozdziału zleceń – BruteForceAlgorithm/BruteForcelgorithm2

Tak jak poprzednio użyte zostały kombinacje powyższych, co dało 12 stanów i 4 akcje.

**Funkcja nagrody:**

$2 * \text{holonCommissions} / \text{holonCommissions} + \text{costOfCommission} / \text{costOfCommission}$

### 3.13.3. Klastrowanie

Jest to metoda dokonująca grupowania elementów we względnie jednorodne klasy.

Zakłada się, że elementy z danej klasy powinny być jak najbardziej podobne do siebie i jak najbardziej różne od elementów z innych klas. Podstawą grupowania jest podobieństwo pomiędzy elementami – wyrażone przy pomocy funkcji (metryki) podobieństwa. Całość polega na podziale zbioru elementów na grupy (klasy, podzbiory). Klastrowanie między innymi służy do zredukowania dużej liczby danych pierwotnych do kilku podstawowych kategorii, które mogą być traktowane jako przedmiot dalszej analizy.

Wyróżniamy trzy podstawowe metody klastrowania:

- Metody hierarchiczne – algorytm tworzy dla zbioru obiektów hierarchię klasyfikacji, zaczynając od takiego podziału, w którym każdy obiekt stanowi samodzielne skupienie, a kończąc na podziale, w którym wszystkie obiekty należą do jednego skupienia
- Metody k-średnich – podział populacji na z góry założoną liczbę klas. Następnie uzyskany podział jest poprawiany w ten sposób, że niektóre elementy są przenoszone do innych klas, tak, aby uzyskać minimalną wariancję wewnętrz uzyskanych klas

- Metody rozmytej analizy skupień - mogą przydzielać element do więcej niż jednej kategorii, stosowane do kategoryzacji (przydziąłu jednostek do jednej lub wielu kategorii)

W systemie użyto metody k-średnich wraz z funkcjami określającymi optymalną ilość klastrów dla danego zbioru.

### 3.13.3.1. Koncepcja

Przed przejściem do implementacji ustalono cele jakie sobie stawiamy oraz wymyślono i przedyskutowano architekturę oraz sposób jej implementacji. Do implementacji wybrano algorytmy klastrujące z rodziny : k-means. W trybie nauki użyto algorytmu pamk, który jest odmianą k-means. Różni się właściwie tylko tym, że w przeciwieństwie do klasycznych k-meansów jako środki klastrów wybierane są elementy wchodzące w skład wektora wejściowego. W dodatku pamk ma bardzo użyteczną cechę – sam wyznacza optymalną ilość klastrów.

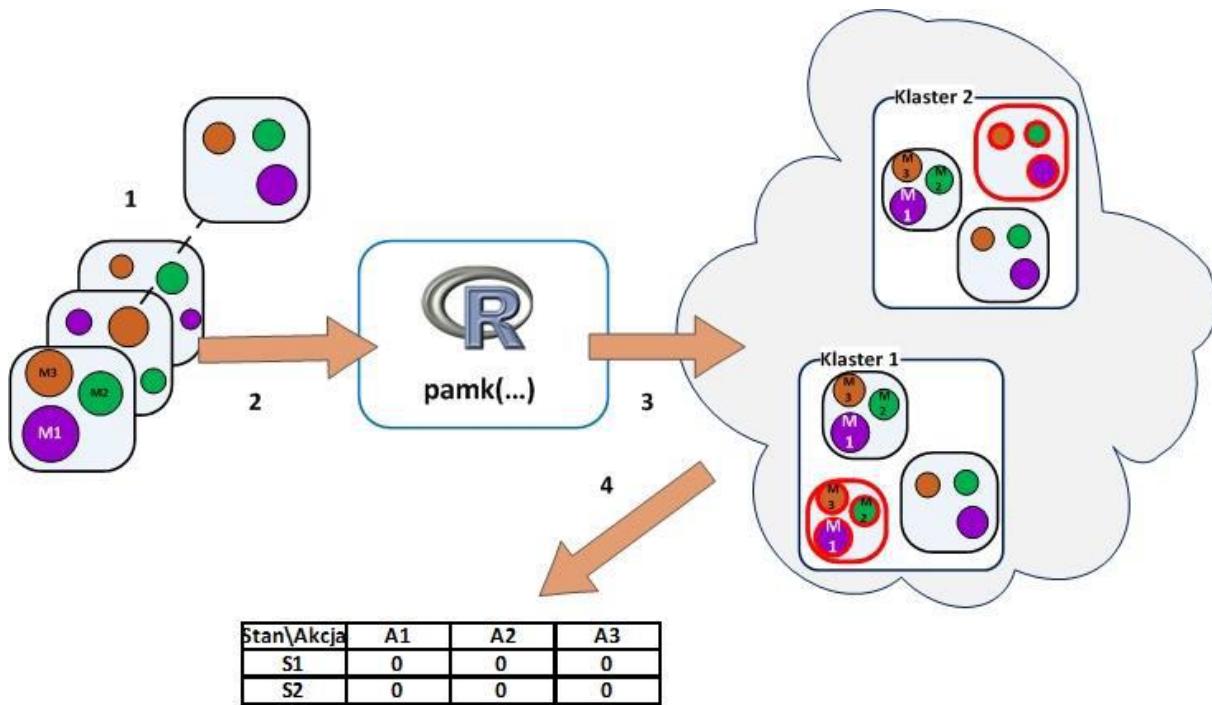
W tym punkcie przedstawimy ogólną koncepcję, która powstała przed przejściem do implementacji.

a) Tryb nauki:

- Wczytujemy konfigurację systemu wraz z wszystkimi stanami oraz zawartością tabeli,
- Stany oraz akcje wybierane są tak samo jak w trybie pracy, jeśli włączona jest eksploracja to komórki mogą się zmieniać,
- W czasie działania zapisywana jest historia measurementów, które bierzemy pod uwagę (konfigurowalne)
- Po skończonej symulacji na podstawie zebranych measurementów przeprowadzana jest klastryzacja, która w efekcie daje możliwe stany,
- Do klastryzacji używana jest metoda pamk, która pozwala na obliczenie optymalnej ilości klastrów, my jedynie ograniczamy tę ilość \*2, pow(ilość\_measurementów, 0.5)+
- Stara konfiguracja, w tym stany oraz zawartość tabeli, jest usuwana
- Konfiguracja z nowymi stanami jest zapisywana do pliku, naszymi stanami są teraz środki klastrów,

- Wartości komórek są w tym wypadku wyzerowane

Spójrzmy na rysunek poglądowy:



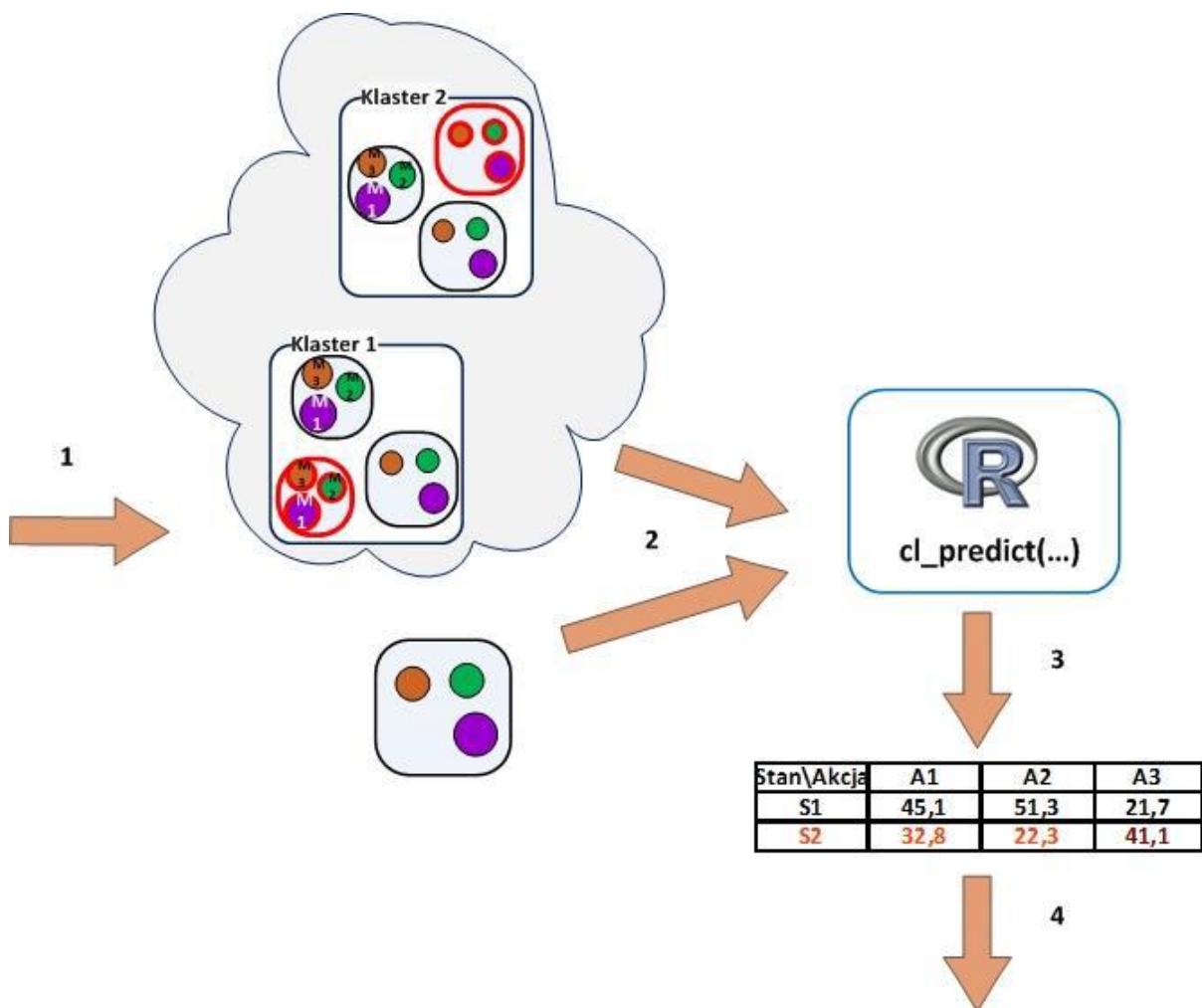
Rysunek 32 Poglądowy rysunek trybu nauczania

- 1) W kolejnych krokach symulacji, zapisywane są wartości measurementów. Jeden wpis w historii, zawiera wartości wszystkich monitorowanych wartości.
- 2) Po skończonej symulacji, zbiór pomiarów jest wykorzystywany do stworzenia klastrów. Są one generowane przez platformę R. Został użyty algorytm pamk – odmiana kmeans.
- 3) Wygenerowane klastry to zbiory pomiarów. Pojedynczy klaster jest reprezentowany przez środek. W przypadku algorytmu pamk, środkiem wybierany jest jeden z punktów klastra.
- 4) Klastry są zapisywane do pliku konfiguracyjnego. Dla Każdego stanu, wartości akcji zostają wyzerowane.

b) Tryb pracy:

- Po każdym kroku measurement jest przyporządkowany do któregoś z istniejących klastrów, czyli wybierany jest stan, zastanawiamy się nad możliwością stworzenia nowego klastra, jeśli rozważany measurement ewidentnie nie pasuje do żadnego z klastrów
- Podobnie jak w Q-learningu stany mogą się zmieniać, na podstawie komórek dla danego stanu wybierane są akcje,
- W trybie eksploracji dochodzi do zmian wartości komórek

Ponownie posłużymy się rysunkiem:



Rysunek 33 Poglądowy rysunek trybu pracy

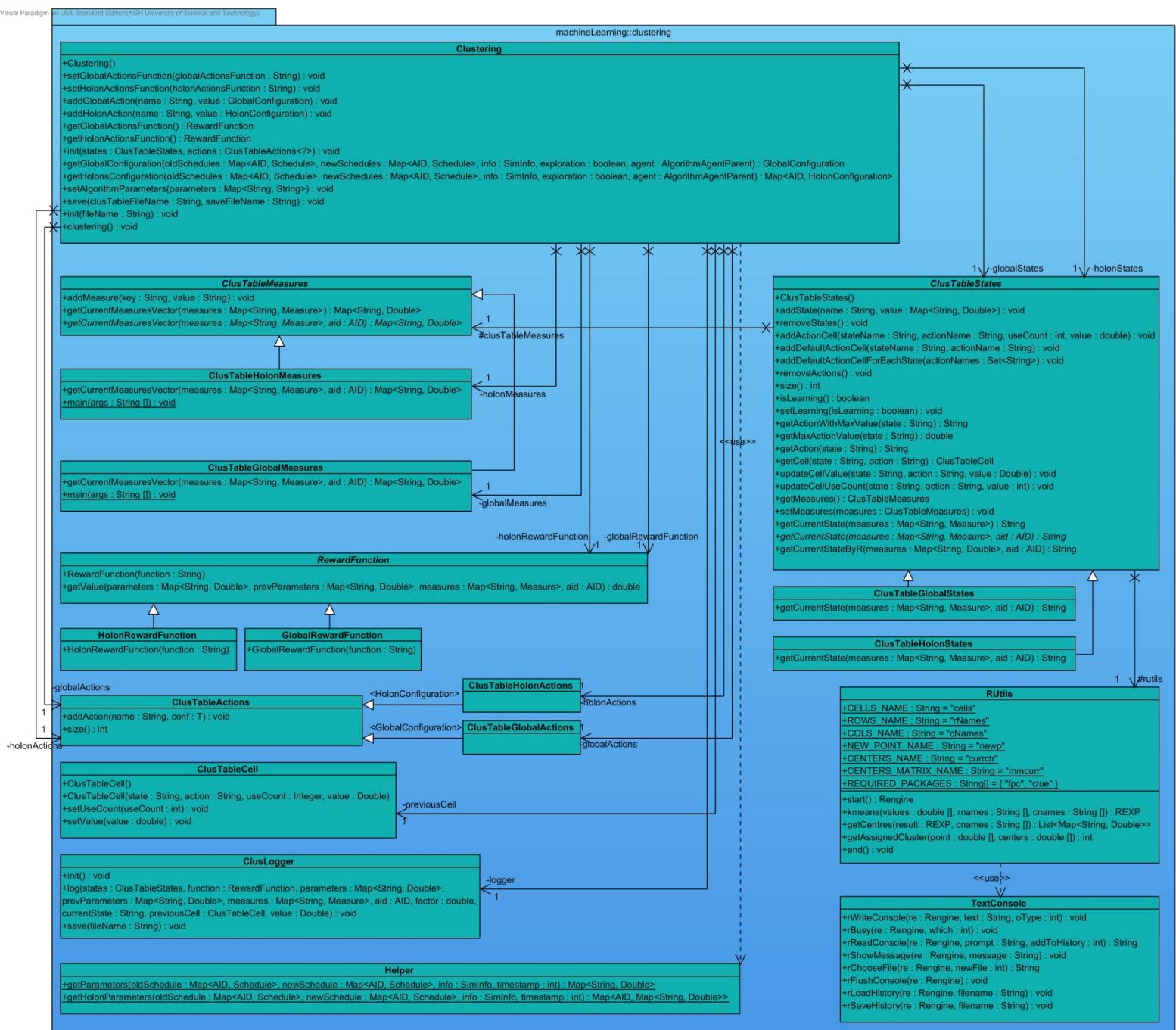
- 1) Wykonuje się krok symulacji. Należy określić stan systemu na podstawie wartości measurementów.

## Dispatch Rider – dokumentacja projektowa

- 2) Ponownie wykorzystana jest platforma R. Na podstawie zbioru klastrów oraz aktualnych pomiarów, funkcja cl\_predict(...) przyporządkowuje punkt do jednego z klastrów.
- 3) Z tabeli utrzymywanie w trakcie pracy wybierany jest stan oraz akcja do wykonania.
- 4) Zaczyna się kolejny krok symulacji

### 3.13.3.2. Diagram klas

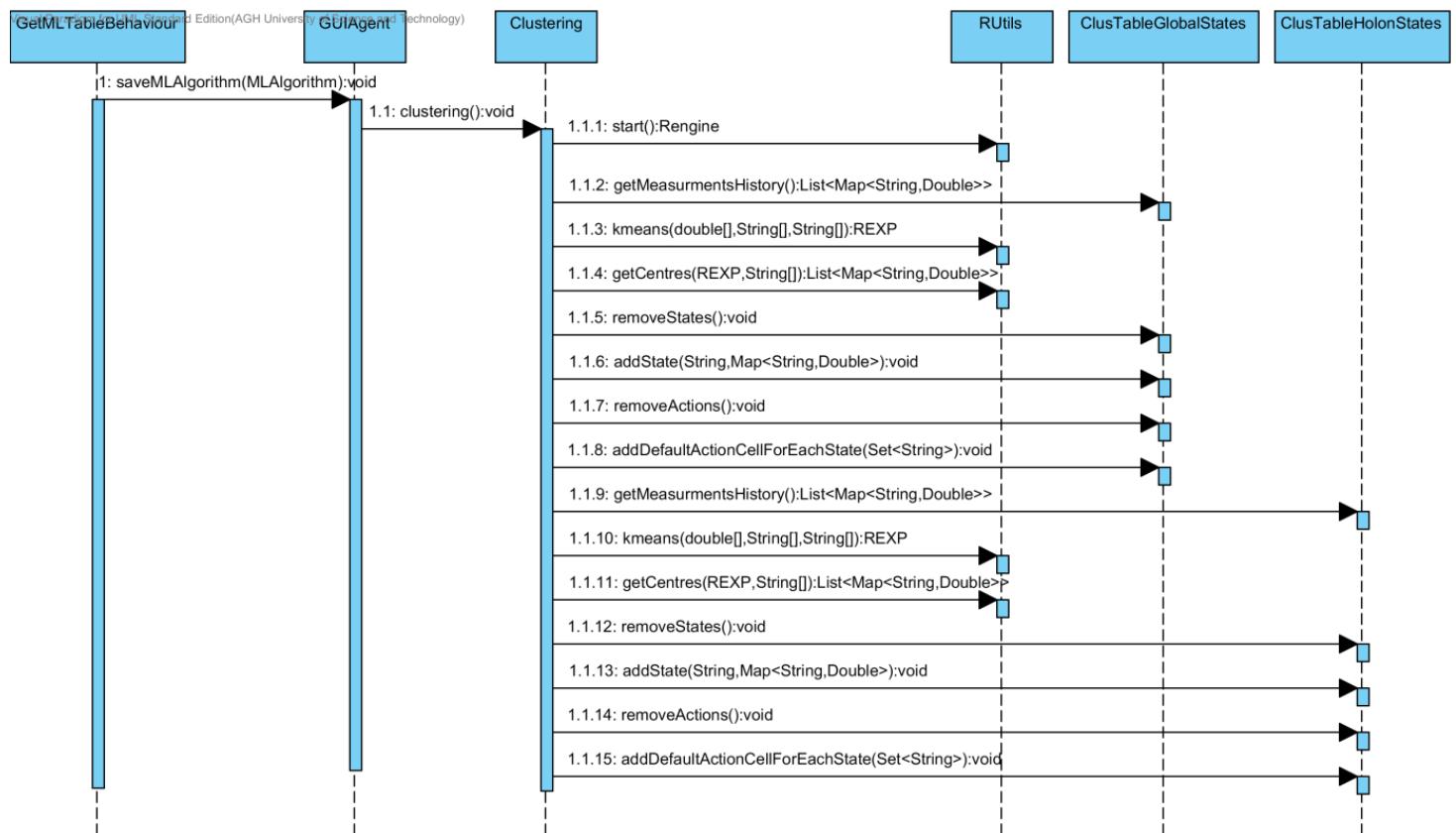
Poniższy diagram klas przedstawia wszystkie klasy zaimplementowane w systemie realizujące Clustering. Podane zostały wszystkie metody oraz zależności między klasami. Przedstawione klasy można znaleźć w pakiecie machineLearning.clustering.



Yrysunek 34 Diagram klas mechanizmu Clusteringu

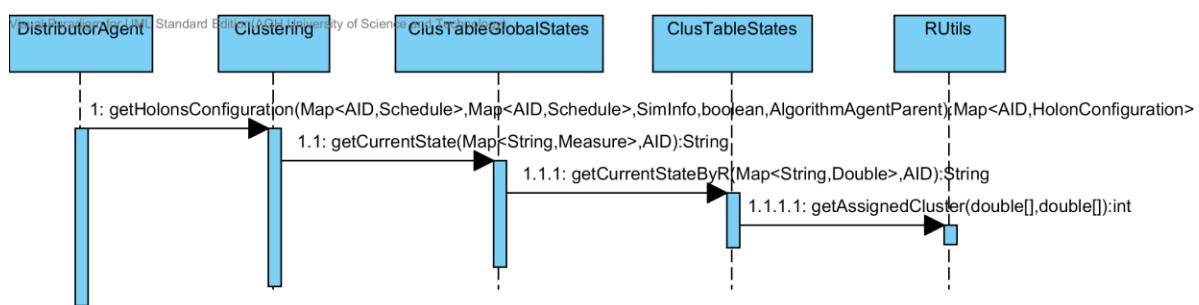
### 3.13.3.3. Diagramy sekwencji

Poniższy diagram sekwencji obrazuje jak dochodzi do zapisania nowego pliku konfiguracyjnego dla Clusteringu. Należy jeszcze raz wspomnieć, że ten proces wywoływany jest tylko w przypadku włączonej opcji learning (parametr learning=true). Przed zapisaniem nowej konfiguracji przeprowadzany jest clustering, cały proces jest dobrze zilustrowany na

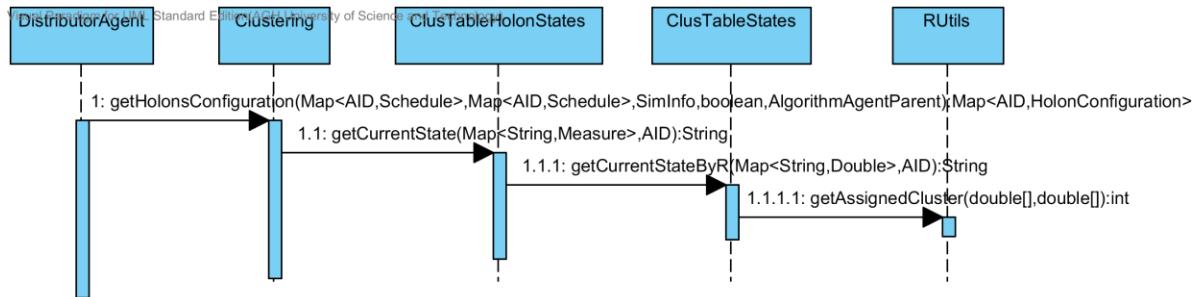


Rysunek 35 Diagram sekwencji - proces zapisu nowego pliku konfiguracyjnego

Poniższe dwa diagramy przedstawiają w jakiej sytuacji używany jest Clustering celem wyboru aktualnego stanu systemu. Pierwszy diagram pokazuje użycie metody cl\_predict (R) w przypadku wyboru konfiguracji globalnej, podczas gdy drugi pokazuje użycie tej metody w przypadku wyboru konfiguracji lokalnej (holonicznej).



Rysunek 36 Diagram sekwencji - użycie metody cl\_predict przy konfiguracji globalnej



Rysunek 37 Diagram sekwencji - użycie metody cl\_predict przy konfiguracji lokalnej (holonicznej)

### 3.13.3.4. Użycie algorytmu w systemie

Aby włączyć uczenie maszynowe z klastrowaniem tak aby było używane w „Dispatch Rider” należy zmienić główną konfigurację : „configuration.xml”. W elemencie mlAlgorithm jako algorithm należy ustawić „Clustering”, jako file – ścieżkę do pliku opisującego tabelę dla klastryzacji, exploration – true gdy zmieniamy wartości tabeli, false gdy nie zmieniamy.

Można również określać dodatkowe parametry (stałe), których możemy potem używać np. w funkcji nagrody.

Dokładny opis znajduje się w instrukcji instalacji i konfiguracji systemu.

### 3.13.3.5. Zapisywanie oraz wczytywanie konfiguracji

W pakiecie machineLearning.xml stworzono dwie klasy. Jedną do wczytywania konfiguracji(*ClutableStructureParser.java*), a drugą do jej zapisywania (*ClutableToXmlWriter.java*).

*ClutableStructureParser* posiada metodę statyczną *parse*, która jest wykonywana na obiekcie *this*(na instancji klasy *Clustering*) przy inicjalizacji algorytmu clusteringu.

*ClutableToXmlWriter* ma statyczną metodę *writeToXml*, która zapisuje *ClusTable* do pliku xml.

Konfiguracja jest zapisywana do pliku w trzech przypadkach:

- learning = true, exploration = true
- learning = true, exploration = false
- learning = false, exploration = true

Wówczas nadpisywany jest plik, który został na początku wczytany, jak również zapisywana jest kopia w folderze, z którego zostały pobrane dane testowe, tag configuration w konfiguracji głównej.

Obie klasy do operacji na XML'ach wykorzystują bibliotekę DOM.

### 3.14. Graf

Aplikacja pozwala na używanie grafu. Wymaganiem jest, żeby używany graf był spójny, tzn. żeby istniała co najmniej jedna ścieżka między dwoma dowolnymi węzłami grafu.

#### 3.14.1. Generacja grafu

Graf o wspomnianej wyżej strukturze można wygenerować używając klasy *dtp.graph.GraphGeneratorTest*. Zostało to opisane w instrukcji konfiguracji i instalacji systemu.

#### 3.14.2. GraphSchedule

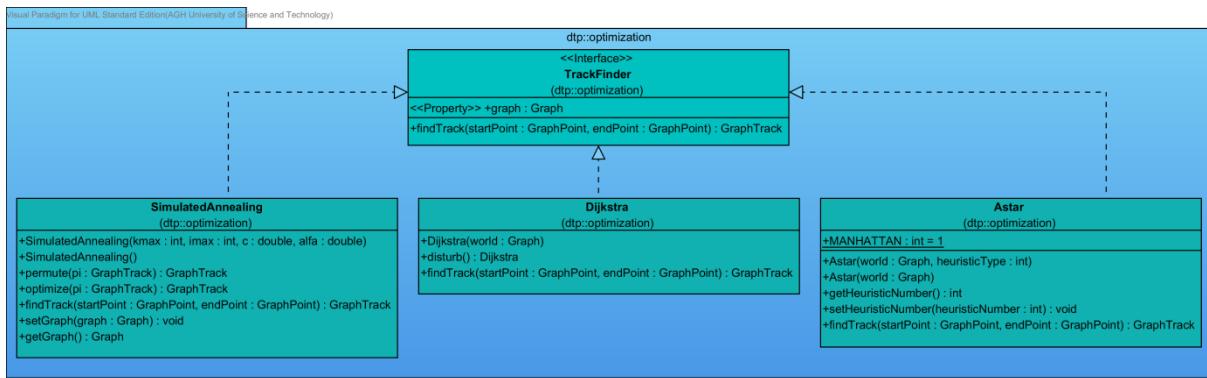
Wprowadzenie grafu spowodowało konieczność zmian w klasie Schedule. Ze względu na to, że wiele elementów się powtarzało w porównaniu do podejścia bez grafu dokonany został mały refactoring, w wyniku którego powstały klasy:

- Klasa abstrakcyjna Schedule ze wspólnymi metodami i polami
- Klasa BasicSchedule – dla reprezentacji bez grafu (zakładamy graf pełny)
- Klasa GraphSchedule – reprezentacja przy używaniu grafu

#### 3.14.3. Wyszukiwanie tras w grafie

W systemie zaimplementowano parę algorytmów służących wyszukiwaniu tras w grafie. Na chwilę obecną wykorzystuje się jedynie algorytmy Dijkstry i A\* (AStar). Konkretny algorytm dla danej symulacji należy wskazać w pliku konfiguracyjnym, co zostało opisane w instrukcji instalacji i konfiguracji systemu. Poniższy diagram obrazuje, jak dodawane są algorytmy wyszukiwania ścieżek do systemu.

# Dispatch Rider – dokumentacja projektowa



Rysunek 38 Diagram klas przedstawiający algorytmy wyszukiwania tras w grafie

## 3.15. Niepewne czasy przejazdu

Po wprowadzeniu grafu przyszedł czas na dodanie zmienności czasów przejazdu między poszczególnymi węzłami w czasie działania symulacji. Odzwierciedla to lepiej realne wymagania występujące w transporcie, takie jak np. korki, spowodowane powrotem ludzi z pracy.

### 3.15.1. Realizacja

Powyższy plik jest reprezentowany w postaci obiektu klasy `dtp.graph.GraphChangesConfiguration`.

W czasie symulacji sprawdzane jest, czy graf musi się zmienić. Jeśli tak, to wysyłana jest informacja o zmianach do wszystkich agentów, którzy korzystają z grafu (Unity i Dystrybutor).

Wprowadzenie zmiennych czasów przejazdu spowodowało powstanie kilku nowych problemów. Powstała konieczność cache'owania poprzednich czasów przed zmianami (ponieważ jeśli jednostka przejechała już jakąś krawędź, to musimy jej czas liczyć po starej wartości). Oprócz tego czas przejazdu może zmieniać się, kiedy jednostka jest w środku krawędzi. Wtedy należy odpowiednio uwzględnić prędkość z jaką jechał do tej pory, oraz nową prędkość (czas przejazdu całej krawędzi). Dodatkowo skomplikował się sam model wyliczania czasu przejazdu z uwagi na to, że z punktu A do B może prowadzić ścieżka przechodząca przez wiele punktów pośrednich (należy je uwzględnić).

Powyższe funkcjonalności są zawarte w `GraphSchedule`.

Na omówienie zasługuje mechanizm wyliczania czasu przejazdu krawędzi w przypadku zmiany jej czasu przejazdu. Wyobraźmy sobie sytuację, że przejeżdżamy z punktu A do B. Początkowo czas pokonania tej trasy wynosi 10, a po 4 jednostkach czasowych zmienia się na 5. Czas przejazdu będzie liczony następująco:

1. Wyznaczana jest prędkość początkowa  $V = \frac{|AB|}{10}$
2. Ponieważ zmiana nastąpiła po 4 jednostkach czasowych, przebyliśmy od punktu A  $V \cdot 4$ , to dystans jaki pozostał do pokonania po zmianie to  $|AB| - 4V$
3. Wyliczamy czas potrzebny na dojście do punktu B na podstawie proporcji:

$$\begin{array}{ccc} |AB| & - & 5 \\ |AB| - 4V & - & x \end{array}$$

$$\text{czyli } x = 5 \frac{|AB| - 4V}{|AB|}$$

4. Dzięki temu możemy wyliczyć czas przejazdu z A do B, który wynosi  $4 + x$

Oczywiście podobnych zmian może być wiele, zanim jednostka dotrze do celu. Wtedy zawsze wystarczy wyznaczyć przejechaną do tej pory część trasy, wyliczyć z proporcji czas dojazdu do celu i dodać do ilości jednostek czasowych, które minęły od wyruszenia z punktu A.

### 3.15.2. Opóźnienia(zaburzenia) w grafie

Wprowadzenie niepewnych czasów przejazdu sprawiło, że powstała konieczność stworzenia generatora opóźnień, który zmieniałby czasy przejazdu między poszczególnymi punktami w grafie.

Generator został zamplementowany w klasie `dtp.graph.GraphChangesGenerator` i został on opisany w instrukcji instalacji i konfiguracji systemu.

Jako parametry przyjmuje:

- `graphFile` – ścieżkę do pliku z opisem grafu, dla którego generujemy plik ze zmianami czasów przejazdu

- commissionsFile – plik z definicją poszczególnych zleceń, na podstawie którego był wygenerowany graf. Praktycznie służy on jedynie pobraniu deadline'u dostarczenia zleceń, żeby wiedzieć, kiedy przestać generować zmiany
- vehicleNr – ilość wirtualnych pojazdów, które zwiększą czas przejazdu w grafie
- minSpeed/maxSpeed – służą do definicji przedziału wartości, o jakie każdy z pojazdów może powiększać czas przejazdu po danej krawędzi grafu
- updateFreq – ilość timestampów, po których ma być zapisywany stan grafu, czyli co ile chcemy zmieniać graf
- Nazwę pliku w metodzie generate – nazwa pod którą ma być zapisany plik ze zmianami czasów przejazdu

Jeśli chodzi o zasadę działania generatora to jest ona następująca:

1. Inicjalizacja wirtualnych pojazdów – parametry każdego pojazdu są losowane na z dopuszczalnych wartości ustawionych podczas wywołania generatora. Są to: czas wyjazdu, o ile pojazd ma pogarszać czas przejazdu w grafie trasa, którą ma pokonać dany pojazd
2. W pętli od 0 do deadline są przemieszczane poszczególne pojazdy – kiedy pojazd wjeżdża na daną krawędź grafu modyfikuje czas, poprzez dodanie swojej prędkości do czasu przejazdu. Jeśli na danej krawędzi znajduje się więcej pojazdów, to czas przejazdu jest modyfikowany tylko przez prędkość pojazdu powodującego największe opóźnienie. Oczywiście kiedy pojazd opuszcza daną krawędź jej czas przejazdu wraca do stanu poprzedniego.
3. Co updateFreq jest zapisywany stan grafu, tzn są zapisywane zmienione czasy przejazdów w poszczególnych krawędziach.

### 3.15.3. Różne czasy dowiadywania się o zmianach w grafie

Oprócz samych zmian w grafie system pozwala na konfigurację momentu, w którym pojazdy dowiadują się o zmianach.

Obecnie system posiada trzy tryby:

- wszystkie pojazdy dowiadują się od razu o wszystkich zmianach (globalnie)
- zmiana jest rozgłoszana dopiero, gdy jakiś pojazd ją wykryje (gdy wjedzie na krawędź ze zmienionym czasem)

- zmiany są rozgłaszone wszystkim, ale po pewnym czasie. Jeśli pojazd wcześniej wykryje zmianę, to ją uwzględnia, nie informując nikogo

#### Konfiguracja:

Konfiguracja powyższego mechanizmu jest bardzo prosta. Wystarczy użyć dwóch nowych atrybutów w elemencie *graph*.

Są to:

- *changeTime* – *immediately/afterChangeNotice/afterTime* – kolejność parametrów odpowiada kolejności w liście wyżej
- *notificationTime* – używane tylko, gdy *changeTime="aftertime"*. Określa po ilu timestampach zmiany mają zostać rozgłoszone wszystkim pojazdom

#### 3.15.4. Algorytmy predykacji zmian w grafie

Dodawanie kolejnych algorytmów jest bardzo proste. Wystarczy tylko dopisać nową klasę i trzymać się ustalonych konwencji nazewniczych.

Nasza klasa musi:

- rozszerzać klasę *dtp.graph.predictor.GraphLinkPredictor*
- znajdować się w pakiecie *dtp.graph.predictor*
- nazwa klasy musi kończyć się na *GraphLinkPredictor*

#### Konfiguracja:

Żeby użyć predyktorów wystarczy użyć dwóch atrybutów w elemencie *graph*:

- *predictor* – jego wartość musi być przedrostkiem nazwy klasy predyktora po obcięciu końcówki *GraphLinkPredictor*. Obecnie są zaimplementowane predyktory: Standard (brak predykcji), Average (średnia), WeightedAverage (0.5\*poprzednia wartość + 0.5\*średnia z jeszcze wcześniejszych wartości), MovingAverage (średnia ruchoma ekspotencjalna)
- *historySize* – ilość pamiętanych grafów (na ich podstawie wyliczane są koszty połączeń w grafie)

#### 3.16. Używanie agentów pomocniczych

System oferuje możliwość zwracania się o policzenie jakiś skomplikowanych, lub długotrwałych obliczeń do oddzielnych agentów pomocniczych.

### 3.16.1. Tworzenie agentów pomocniczych

W celu stworzenia (implementacji) nowego agenta pomocniczego należy stworzyć nową klasę, która rozszerza `dtp.jade.algorithm.agent.AlgorithmAgent`

Musimy wtedy zaimplementować dwie metody:

- `init` – jako parametr podawane są wartości inicjalizacyjne ustawiane w konfiguracji testu. Są tam parametry dla wszystkich agentów pomocniczych, dlatego powinniśmy używać tylko tych, które są przeznaczone dla naszego agenta
- `getResponse` – metoda, która prosi agenta o wykonanie jakiejś operacji, której nazwa jest w parametrze `operation`, natomiast jej parametry są w argumencie `params`

### 3.16.2. Konfigurowanie agentów pomocniczych

Służy do tego element `algorithmAgentsConfig` i został on opisany w instrukcji instalacji i konfiguracji systemu.

### 3.16.3. Używanie agentów pomocniczych

Agentów pomocniczych możemy używać w dwóch miejscach:

- Klasy dziedziczące po `algorithm.Algorithm`
- Klasy dziedziczące po `algorithm.comparator.CommissionsComparator`

W tym celu każda z naszych klas musi zwracać listę klas naszych agentów pomocniczych w metodzie `getHelperAgentsClasses`. Tylko Ci agenci będą stworzeni i dostępni w czasie symulacji. Następnie w dowolnym momencie możemy wywołać metodę `askAgent`, która zwraca odpowiedź od agenta pomocniczego, a jako parametry przyjmuje:

- Klasę agenta pomocniczego (jedną z tych, które są zwracane w metodzie `getHelperAgentClasses`).
- Nazwę operacji, którą agent ma wykonać
- Parametry potrzebne do jej wykonania
- Agenta, który może komunikować się z agentem pomocniczym (najczęściej jest to `Distributor` lub `Eunit`)

### 3.17. Komparatory zleceń przyznawanych przez Dystrybutora

Obecnie jest możliwość konfigurowania odpowiedniego komparatora, który sortuje zlecenia (które mają zostać rozdzielone w danym timestamp'ie), przed rozpoczęciem ich dystrybucji.

#### 3.17.1. Tworzenie nowych komparatorów

W celu stworzenia nowego komparatora należy:

- Stworzyć nową klasę, należącą do pakietu algorithm.comparator
- Jej nazwa musi kończyć się na CommissionsComparator
- Musi dziedziczyć po klasie algorithm.comparator.CommissionsComparator

#### 3.17.2. Konfigurowanie użycia komparatora

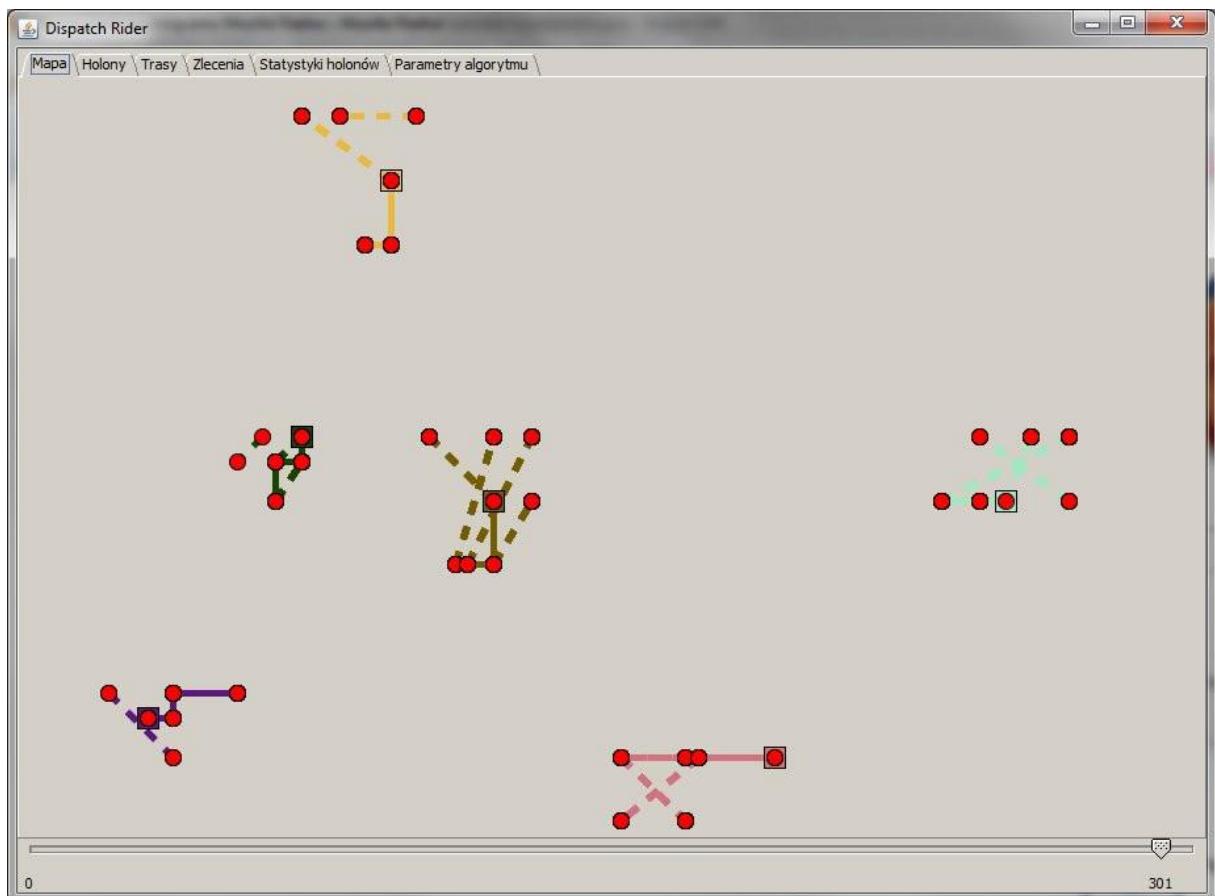
Żeby włączyć w konfiguracji użycie odpowiedniego komparatora, należy użyć atrybutu commissionsComparator w elemencie commissions. Trzeba ustawić jego wartość na nazwę naszego komparatora po odjęciu końcówki CommissionsComparator. Dokładna konfiguracja została opisana w instrukcji instalacji i konfiguracji systemu.

### 3.18. Interfejs GUI

Interfejs GUI umożliwia w czasie zbliżonym do rzeczywistego obserwować efekty pracy agentów Dispatch Ridera. Po dojściu do odpowiedniego momentu w obliczeniach (tzw. timestamp), dane zostają udostępnione użytkownikowi.

#### 3.18.1. Suwak czasu

Do przemieszczania się pomiędzy różnymi momentami czasowymi służy suwak znajdujący się u dołu okna. Suwak został zaprojektowany tak, aby umożliwiać wygodne i intuicyjne obserwowanie interesujących danych. Gdy jest ustawiony na ostatni dostępny timestamp automatycznie przeskakuje na kolejny, gdy ten zostanie przesłany. W innym przypadku suwak pozostanie na swoim miejscu do czasu interakcji użytkownika. Zapobiega to “przeskakiwaniu” danych w nieoczekiwany momencie.



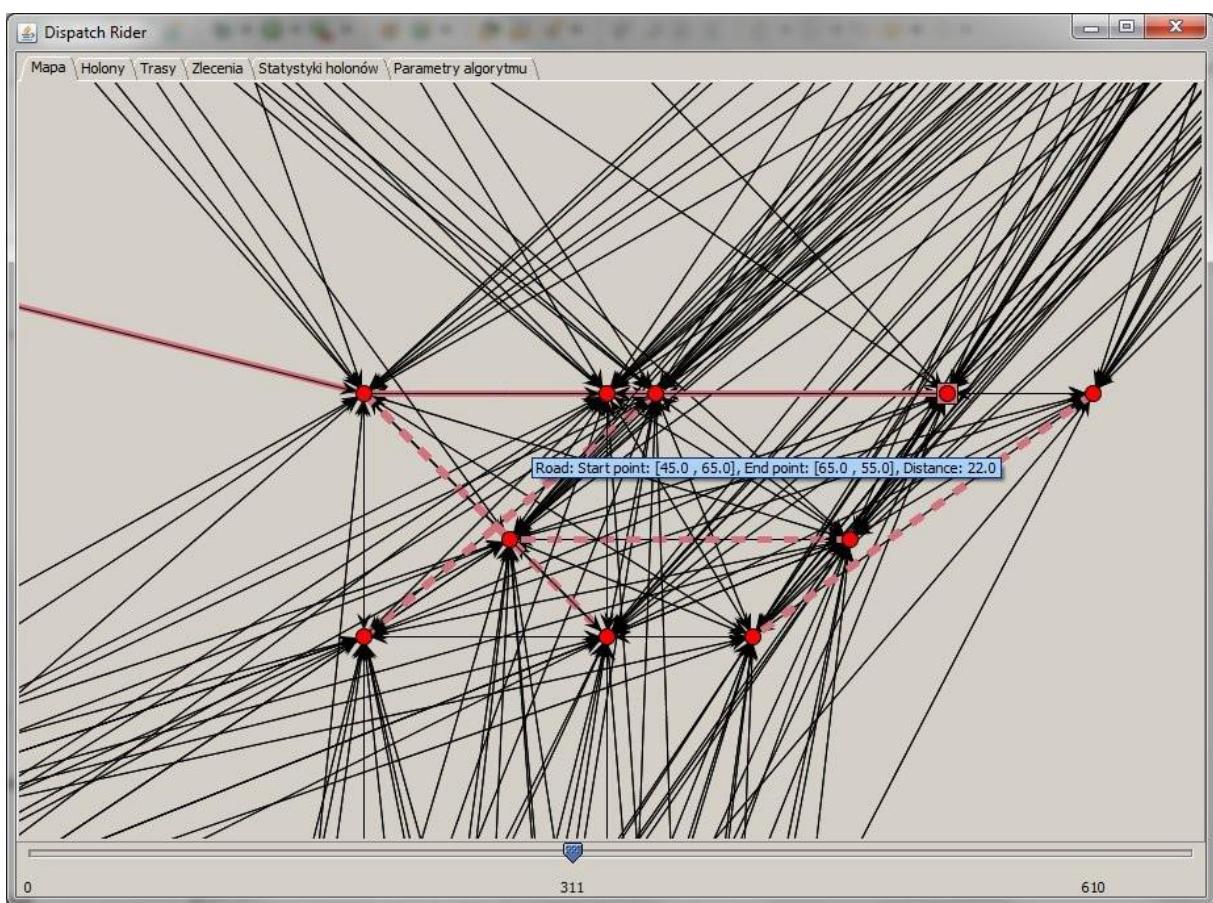
Rysunek 39 Suwak czasu

### 3.18.2. Zakładki

Do przemieszczania się pomiędzy wizualizowanymi danymi służą zakładki. Zakładki pozwalają na szybkie dotarcie do interesujących nas danych poprzez tematycznych ich podział.

### 3.18.2.1. Zakładka „Mapy”

Mapa obrazuje rozmieszczenie w przestrzeni kluczowych punktów - holonów, punktów końcowych zleceń. Wskazanie myszką konkretnego elementu prowadzi do wyświetlenia istotnych elementów w postaci tzw. tooltipa. Wyświetlanie jest w pełni konfigurowalne przy pomocy myszki. Za pomocą scrolla można widok przybliżać i oddalać, przy pomocy lewego przycisku można go przesuwać. Linia ciągła reprezentuje zlecenia wykonywane do danego momentu czasowego. Linia przerywana reprezentuje zlecenia planowane w danym momencie czasowym.



Rysunek 40 Zakładka Mapa

W skład tooltipa wchodzą:

<b>ID</b>	ID Holona
<b>Creation time</b>	Czas utworzenia
<b>Coordinates</b>	Pozycja na mapie
<b>Truck comfort</b>	Komfort pojazdu
<b>Trailer capacity</b>	Pojemność przyczepy

Driver	Kierowca
--------	----------

W skład tooltipa krawędzi wchodzą punkt początkowy, końcowy oraz dystans, a w tooltipie węzła znajdują się jego ID i współrzędne.

### 3.18.2.2. Zakładki poszczególnych tabel

Pozostałe zakładki wyświetlają parametry wyliczone przez system dla poszczególnych holonów i zleceń w formie tabel. Dane można łatwo segregować sortując je po wartościach w kolumnie, bądź modyfikując kolejność kolumn.

HolonID	Creation time	Location	Driver name	Truck name
0	0	(85.0, 35.0)	Driver #0	Truck #0
1	0	(60.0, 80.0)	Driver #10	Truck #10
2	0	(42.0, 15.0)	Driver #11	Truck #2
3	0	(8.0, 40.0)	Driver #12	Truck #12
4	0	(18.0, 75.0)	Driver #13	Truck #14
5	0	(65.0, 55.0)	Driver #14	Truck #18
6	0	(48.0, 40.0)	Driver #15	Truck #5
7	0	(33.0, 32.0)	Driver #16	Truck #16
8	0	(42.0, 66.0)	Driver #17	Truck #3
9	0	(25.0, 50.0)	Driver #18	Truck #8

Rysunek 41 Zakładka Holony

W skład tabeli holonów wchodzą:

HolonID	ID holona
Creation time	Czas utworzenia
Location	Pozycja na mapie
Driver name	Nazwa kierowcy
Truck name	Nazwa ciężarówki

CommisionID	Holon ID	Pickup ID	Pickup X	Pickup Y	Pickup Time 1	Pickup Time 2	PickupServiceTime	Delivery ID	Delivery X	Delivery Y	Delivery Time 1	Delivery Time 2
0	8	3	42.0	66.0	65.0	146.0	75	45.0	65.0	997.0	1068.0	
1	8	5	42.0	65.0	15.0	67.0	7	40.0	66.0	170.0	225.0	
3	8	38	68.0	255.0	324.0	10	35.0	66.0	357.0	410.0		
6	4	13	22.0	75.0	30.0	92.0	17	18.0	75.0	99.0	148.0	
8	4	18	15.0	75.0	179.0	254.0	12	25.0	85.0	652.0	721.0	
9	4	19	15.0	80.0	278.0	345.0	15	20.0	80.0	384.0	429.0	
10	9	20	30.0	50.0	10.0	73.0	24	25.0	50.0	65.0	144.0	
12	9	25	25.0	52.0	169.0	224.0	27	23.0	52.0	261.0	316.0	
17	3	33	8.0	40.0	87.0	158.0	37	2.0	40.0	383.0	434.0	
16	3	32	10.0	40.0	31.0	100.0	31	10.0	35.0	200.0	237.0	
18	3	35	5.0	35.0	283.0	344.0	39	0.0	45.0	567.0	624.0	
21	7	42	33.0	32.0	68.0	149.0	40	35.0	30.0	264.0	321.0	
22	7	43	33.0	35.0	16.0	80.0	41	35.0	32.0	166.0	235.0	
29	2	54	42.0	10.0	186.0	257.0	60	35.0	5.0	562.0	629.0	
28	2	53	44.0	5.0	286.0	347.0	58	38.0	5.0	471.0	534.0	
31	2	57	40.0	15.0	35.0	87.0	55	42.0	15.0	95.0	158.0	
35	6	65	48.0	40.0	76.0	129.0	72	53.0	30.0	450.0	505.0	
32	6	62	50.0	35.0	262.0	317.0	68	45.0	30.0	734.0	777.0	
33	6	63	50.0	40.0	171.0	218.0	74	53.0	35.0	353.0	412.0	
38	0	71	95.0	35.0	293.0	360.0	77	88.0	30.0	574.0	643.0	
39	0	76	90.0	35.0	203.0	260.0	73	92.0	30.0	478.0	551.0	
37	6	67	47.0	40.0	12.0	77.0	61	50.0	30.0	531.0	610.0	
42	0	81	85.0	35.0	47.0	124.0	70	95.0	30.0	387.0	456.0	
40	0	78	88.0	35.0	109.0	170.0	104	88.0	35.0	109.0	170.0	
46	5	87	65.0	55.0	85.0	144.0	83	72.0	55.0	265.0	338.0	
47	5	90	60.0	55.0	20.0	84.0	88	65.0	60.0	645.0	708.0	

Rysunek 42 Zakładka Zlecenia

W skład tabeli zleceń wchodzą:

<b>CommissionID</b>	ID zlecenia
<b>Holon ID</b>	ID holonu
<b>Pickup ID</b>	ID załadowania towaru
<b>Pickup X</b>	Współrzędna x miejsca załadowania
<b>Pickup Y</b>	Współrzędna y miejsca załadowania
<b>Pickup Time 1</b>	Czas początkowy załadowania
<b>Pickup Time 2</b>	Czas końcowy załadowania
<b>PickUpServiceTime</b>	Czas obsługi załadowania
<b>Delivery ID</b>	ID rozładunku
<b>Delivery X</b>	Współrzędna x miejsca rozładunku
<b>Delivery Y</b>	Współrzędna y miejsca rozładunku
<b>Delivery Time 1</b>	Czas początkowy rozładunku
<b>Delivery Time 2</b>	Czas końcowy rozładunku

## Dispatch Rider – dokumentacja projektowa

<b>PickUpDeliveryTime</b>	Czas obsługi rozładunku
<b>Load</b>	Załadunek
<b>ActualLoad</b>	Rzeczywisty załadunek

Dispatch Rider

Mapa \ Holony \ Zlecenia \ **Statystyki holonów** \ Parametry algorytmu \

HolonID	Summary Cost	Cost	DriveTime	SummaryPunishment	Time	WaitTime
0	538.0	123.0	123.0	0.0	538.0	145.0
1	442.0	82.0	82.0	0.0	442.0	0.0
2	253.0	73.0	73.0	0.0	253.0	0.0
3	513.0	91.0	91.0	0.0	513.0	62.0
4	251.0	71.0	71.0	0.0	251.0	0.0
5	762.0	67.0	67.0	0.0	762.0	335.0
6	654.0	47.0	47.0	0.0	654.0	247.0
7	404.0	44.0	44.0	0.0	404.0	0.0
8	1102.0	38.0	38.0	0.0	1102.0	704.0
9	210.0	30.0	30.0	0.0	210.0	0.0

Rysunek 43 Zakładka Statystyki holonów

W skład tabeli statystyk holonów wchodzą:

<b>HolonID</b>	ID holonu
<b>Summary Cost</b>	Koszt sumaryczny
<b>Cost</b>	Koszt
<b>DriveTime</b>	Czas transportu
<b>SummaryPunishment</b>	Sumaryczna kara
<b>Time</b>	Czas sumaryczny
<b>WaitTime</b>	Czas postoju

Dispatch Rider

Mapa \ Holony \ Zlecenia \ **Statystyki holonów** \ Parametry algorytmu \

timestamp	choosingB...	commi...	dist	simmulatedTra...	chooseWorstC...	algorithm	maxFullSTDepth	STTimestampGap	STCommissions...
0	1	8	wTime	1	1	algorithm.BruteForceAlg...	true	false	true

Rysunek 44 Zakładka Parametry algorytmu

W skład tabeli parametrów algorytmu wchodzą:

<b>timestamp</b>	Punkt czasowy
<b>choosingByCost</b>	Tryb przydzielania zleceń
<b>commissionSendingType</b>	Sposób wysyłania zleceń (pojedynczo/grupami)
<b>dist</b>	Sposób wyliczania kosztu zlecenia (czas/dystans)
<b>simmulatedTradingCount</b>	Ilość uruchomień Simulated Trading
<b>chooseWorstCommission</b>	Sposób wyboru najgorszego zlecenia
<b>algorithm</b>	Rodzaj algorytmu rozdziału zleceń
<b>maxFullISTDepth</b>	Maksymalny stopień zagłębiania w algorytmie complexST
<b>STTimestampGap</b>	Brak opisu w udostępnionej dokumentacji
<b>STCommissionsGap</b>	Brak opisu w udostępnionej dokumentacji

Dispatch Rider														
Mapa \ Holony \ Trasy \ Zlecenia \ Statystyki holonów \ Parametry algorytmu														
0	timestamp	status	holon location	commission ID	pickup location	delivery location	pickup time 1	pickup time 2	delivery time 1	delivery time 2	pickup id	delivery id	actual load	load
0	EN ROUTE	(40,0, 50,0)	42	(85,0, 35,0)	(95,0, 30,0)	47,0	124,0	387,0	456,0	70	81	0,0	30	
7	EN ROUTE	(44.7894736842... 42		(85,0, 35,0)	(95,0, 30,0)	47,0	124,0	387,0	456,0	70	81	0,0	30	
10	EN ROUTE	(46.8421052631... 42		(85,0, 35,0)	(95,0, 30,0)	47,0	124,0	387,0	456,0	70	81	0,0	30	
18	EN ROUTE	(52.3157894736... 42		(85,0, 35,0)	(95,0, 30,0)	47,0	124,0	387,0	456,0	70	81	0,0	30	
27	EN ROUTE	(61,0, 35,0)	42	(85,0, 35,0)	(95,0, 30,0)	47,0	124,0	387,0	456,0	70	81	0,0	30	
29	EN ROUTE	(63,0, 35,0)	42	(85,0, 35,0)	(95,0, 30,0)	47,0	124,0	387,0	456,0	70	81	0,0	30	
37	EN ROUTE	(71,0, 35,0)	42	(85,0, 35,0)	(95,0, 30,0)	47,0	124,0	387,0	456,0	70	81	0,0	30	
51	PICKUP	(85,0, 35,0)	42	(85,0, 35,0)	(95,0, 30,0)	47,0	124,0	387,0	456,0	70	81	0,0	30	
111	PICKUP	(85,0, 35,0)	42	(85,0, 35,0)	(95,0, 30,0)	47,0	124,0	387,0	456,0	70	81	0,0	30	
113	PICKUP	(85,0, 35,0)	42	(85,0, 35,0)	(95,0, 30,0)	47,0	124,0	387,0	456,0	70	81	0,0	30	
115	PICKUP	(85,0, 35,0)	42	(85,0, 35,0)	(95,0, 30,0)	47,0	124,0	387,0	456,0	70	81	0,0	30	
121	PICKUP	(85,0, 35,0)	42	(85,0, 35,0)	(95,0, 30,0)	47,0	124,0	387,0	456,0	70	81	0,0	30	
128	PICKUP	(85,0, 35,0)	42	(85,0, 35,0)	(95,0, 30,0)	47,0	124,0	387,0	456,0	70	81	0,0	30	
145	PICKUP/DELIVERY	(88,0, 35,0)	40	(88,0, 35,0)	(88,0, 35,0)	109,0	170,0	109,0	170,0	104	78	0,0	20	
197	PICKUP/DELIVERY	(88,0, 35,0)	40	(88,0, 35,0)	(88,0, 35,0)	109,0	170,0	109,0	170,0	104	78	0,0	20	
204	PICKUP/DELIVERY	(88,0, 35,0)	40	(88,0, 35,0)	(88,0, 35,0)	109,0	170,0	109,0	170,0	104	78	0,0	20	
220	PICKUP/DELIVERY	(88,0, 35,0)	40	(88,0, 35,0)	(88,0, 35,0)	109,0	170,0	109,0	170,0	104	78	0,0	20	
225	PICKUP/DELIVERY	(88,0, 35,0)	40	(88,0, 35,0)	(88,0, 35,0)	109,0	170,0	109,0	170,0	104	78	0,0	20	
228	PICKUP/DELIVERY	(88,0, 35,0)	40	(88,0, 35,0)	(88,0, 35,0)	109,0	170,0	109,0	170,0	104	78	0,0	20	
235	EN ROUTE	(88,0, 35,0)	39	(90,0, 35,0)	(92,0, 30,0)	203,0	260,0	478,0	551,0	73	76	0,0	10	
300	PICKUP	(90,0, 35,0)	39	(90,0, 35,0)	(92,0, 30,0)	203,0	260,0	478,0	551,0	73	76	0,0	10	
301	PICKUP	(90,0, 35,0)	39	(90,0, 35,0)	(92,0, 30,0)	203,0	260,0	478,0	551,0	73	76	0,0	10	
310	PICKUP	(90,0, 35,0)	39	(90,0, 35,0)	(92,0, 30,0)	203,0	260,0	478,0	551,0	73	76	0,0	10	
311	PICKUP	(90,0, 35,0)	39	(90,0, 35,0)	(92,0, 30,0)	203,0	260,0	478,0	551,0	73	76	0,0	10	
327	EN ROUTE	(90,0, 35,0)	38	(95,0, 35,0)	(88,0, 30,0)	293,0	360,0	574,0	643,0	77	71	0,0	20	
390	PICKUP	(95,0, 35,0)	38	(95,0, 35,0)	(88,0, 30,0)	293,0	360,0	574,0	643,0	77	71	0,0	20	

Rysunek 45 Zakładka Trasy

Zakładka tras pozwala wyświetlić wygenerowaną trasę dla danego holonu. W celu wskazania należy wybrać odpowiednie ID z rozwijalnej listy umieszczonej nad tabelą.

W skład tabeli trasy wchodzą:

<b>timestamp</b>	Punkt czasowy
<b>status</b>	EN ROUTE – w drodze, PICKUP – trwa odbiór towaru, DELIVERY – trwa dostarczanie towaru
<b>holon location</b>	Pozycja holonu na mapie
<b>Commission ID</b>	ID aktualnie realizowanego zlecenia
<b>pickup location</b>	Miejsce załadunku
<b>delivery location</b>	Miejsce rozładunku
<b>pickup time 1</b>	Czas początkowy załadunku
<b>pickup time 2</b>	Czas końcowy załadunku
<b>delivery time 1</b>	Czas początkowy rozładunku
<b>delivery time 2</b>	Czas końcowy rozładunku
<b>pickup ID</b>	ID załadunku
<b>delivery ID</b>	ID rozładunku
<b>actual load</b>	Rzeczywisty załadunek
<b>load</b>	Załadunek

### 3.18.3. Zmiany wprowadzone w systemie

W celu połączenia modułu GUI z resztą systemu, należało dokonać kilku modyfikacji.

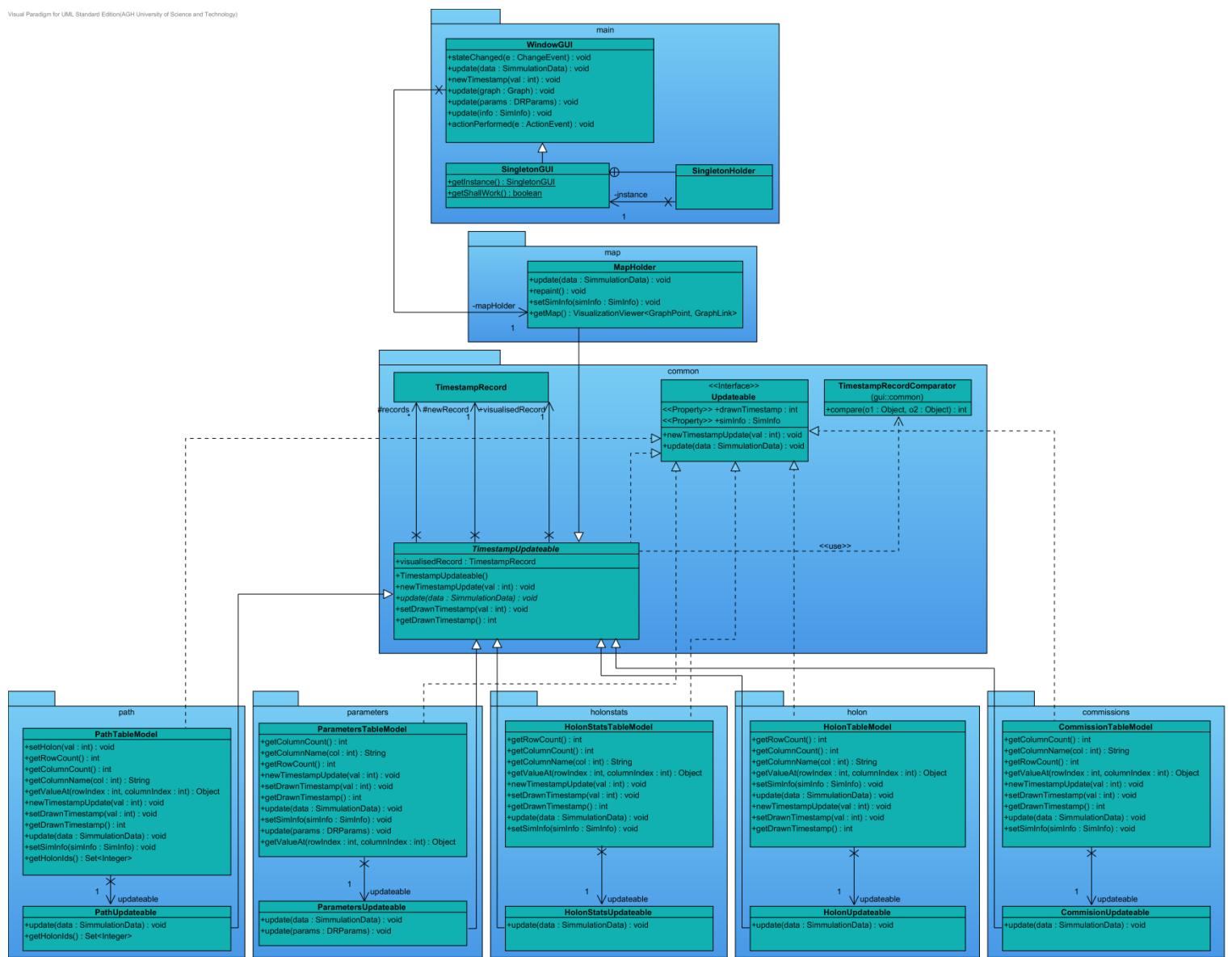
Drobnym zmianom uległy metody:

- `dtp.jade.gui.GUIAgent.addSimmulationData`
- `dtp.jade.gui.GUIAgent.changeGraph`
- `dtp.jade.gui.GUIAgent.simulationStart`
- `dtp.jade.gui.GUIAgent.sendTimestamp`
- `dtp.jade.distributor.DistributorAgent.setCommissions`

Dodano w nich kod przesyłający dane prezentowane użytkownikowi. Instrukcje te są postaci `SingletonGUI.getInstance().metoda`.

### 3.18.4. Architektura GUI

Visual Paradigm for UML, Standard Edition/AGH University of Science and Technology



Rysunek 46 Diagram klas mechanizmu GUI

#### 3.18.4.1. Pakiet `gui.main`

Jest to pakiet w którym znajdują się 2 klasy. Są to prosta `SingletonGUI`

implementująca jedynie wzorzec singleton i dziedzicząca po drugiej klasie, `WindowGUI`, które to jest jedynym punktem dostępu do naszego modułu z zewnątrz, a zarazem podstawową klasą interfejsu użytkownika. W klasie `WindowGUI` zdefiniowane są wszystkie metody wspomniane w "Zmianach w dostarczonym kodzie", służące do pobierania danych z właściwego systemu DispatchRider. Metody te informują z kolei klasy odpowiadające za poszczególne panele (zakładki) o zachodzących zmianach.

Jeśli chodzi o elementy widoczne dla użytkownika, cała nasza aplikacja, a co za tym idzie klasa WindowGUI napisana jest z użyciem swinga. WindowGUI jako klasa spinająca cały interfejs odpowiada za poprawne wyświetlenie zakładek, odpowiednie przeskakowanie ich zawartości, a także za widoczny we wszystkich zakładkach suwak (oś czasu) i poprawne jego zachowanie, spięcie z wyświetlonymi danymi.

#### **3.18.4.2. Pakiet *gui.common***

Pakiet ten służy uspójnieniu struktury klas i metod odpowiadających za wyświetlanie danych w poszczególnych zakładkach. Zawiera klasę TimestampRecord mającą w założeniu przechowywać dowolny zestaw danych potrzebny w danej zakładce, ale danych powiązanych z danym momentem czasowym (np. aktualna pozycja holonów w danym czasie w przypadku mapy). Klasa ta ma utworzony swój Comparator (TimestampRecordComparator), konieczny do poprawnego zachowania suwaka czasu. Pozostałe 2 klasy znajdujące się w tym pakiecie to interfejs Updateable, który implementują klasy odpowiadające za poszczególne zakładki, a także abstrakcyjną klasę TimestampUpdateable, w której przechowywane są implementacje tych metod, których wykonanie powinno być takie samo dla wszystkich zakładek (w celu uniknięcia kopiowania tego samego kodu w kilku miejscach).

#### **3.18.4.3. Pakiet *gui.map***

Pakiet ten ze względu na odmienną treść wyświetlana na ekran odbiega znacznie strukturą od pozostałych pakietów odpowiadających za dane w zakładkach. Podstawową klasą notyfikowaną o wszelkich zdarzeniach jest klasa MapHolder. Wywołanie na niej metody repaint() (odbywa się to z poziomu WindowGUI) powoduje odświeżenie zawartości wyświetlonej mapy. Klasa ta posiada również standardowy zestaw metod do pobierania danych do symulacji. Pozostałe metody, w głównej mierze prywatne, odpowiadają za wyświetlanie poszczególnych elementów. Ich nazwy są bardzo intuicyjne, a zadania wyraźnie rozdzielone. Są to wstawienie wierzchołków grafów, holonów, krawędzi, ustawienie wszelkich właściwości takich jak grubość linii czy kształt strzałek itd. Pozostałe klasy z tego pakietu to HolonGraphLink i HolonGraphPoint służące do rozróżnienia przez nas krawędzi i wierzchołków “standardowych” od tych związanych z przebytą trasą czy holonem. W

pakiecie tym znajdziemy również klasę InvisibleGraphPoint ułatwiającą nam ładne rysowanie przebytej przez holon trasy, a także klasę ColorCreator, której jedynym zadaniem jest stworzenie możliwie różnych kolorów dla wszystkich holonów.

#### **3.18.4.4. Pakiet *gui.commissions, gui.holon, gui.holonstats, gui.parameters, gui.path***

Wszystkie te pakiety są bardzo zbliżone do siebie strukturą i rolą. Każda z zakładek, za które odpowiadają wyświetla pojedynczą tabelę. Poszczególne pakiety posiadają własną klasę \*Updateable rozszerzającą TimestampUpdateable, w której to zaimplementowana jest metoda update (SimmulationData). Rolą tych klas jest dopasowanie przechowywanych danych do tego, co potrzeba w tabeli wyświetlić. Klasy \*TableModel odpowiadają z kolei już za same tabele i odpowiednie zgranie ich z suwakiem czasowym.

## **4. Opis sposobu wprowadzania niektórych zmian**

Rozdział zawiera wytyczne do tego w jaki sposób można by było wprowadzić różnego rodzaju mechanizmy. Przy każdym z nich są też odnośniki do miejsc w kodzie, gdzie dana implementacja powinna się znaleźć.

### **4.9. Zmiany konfiguracji – dodawanie nowych parametrów konfiguracyjnych**

Ponieważ w większości nowych mechanizmów występuje konieczność wprowadzenia nowych parametrów konfiguracyjnych, poniżej znajduje się opis jak to zrobić:

1. modyfikujemy *configuration.xsd* – do schemu należy dodać definicje poszczególnych nowych elementów/atributów
2. dodajemy parsowanie w *dtp.xml.ConfigurationParser*
3. wyniki parsingu zapisujemy w klasie *dtp.jade.test.TestConfiguration* – wystarczy dodać odpowiednie pole + gettery i settery
4. w celu przekazania parametrów do aplikacji musimy je pobrać z *TestConfiguration* w *dtp.jade.test.TestAgent* – metoda *next2*
5. teraz trzeba parametry przekazać do *dtp.jade.gui.GuiAgent* – jest to klasa bazowa *TestAgent*. Zwyczajnie przepisujemy wartości parametrów do nowo stworzonych pól

6. parametry muszą być często dostarczone do wszystkich agentów, dlatego też są one przesyłane w obiekcie *dtp.simmulation.SimInfo* – musimy tam dodać odpowiednie pola, po czym je zainicjalizować wartościami w metodach – *sendSimInfo* oraz *sendSimInfoToAll* w GUIAgencie

To wszystko, teraz wystarczy w poszczególnych agentach odwołać się do tych parametrów poprzez obiekt SimInfo.

#### **4.10. Dodanie możliwości, żeby pojazd wyjeżdżał najpóźniej jak się da**

Obecnie podejście jest inne. Pojazdy wyjeżdżają najwcześniej jak mogą.

Wprowadzenie wspomnianego mechanizmu nie jest banalne. Co należy zrobić:

1. Trzeba zmienić implementację *algorithm.Schedule* – są tam metody, które służą wyliczaniu między innymi czasu realizacji zlecenia. Będą to praktycznie wszystkie metody Schedule, które korzystają z metody *calculateTime*.
2. Należyoby w nich dopisać opóźnienie, które byłoby kalkulowane na podstawie dotychczas przydzielonych zleceń (do których oczywiście mamy dostęp) - czasochłonne
3. Obliczanie opóźnienia – oczywiście zależy ono od stosowanego trybu (miękkie czy twardie okna)
  - Twarde okna – trzeba by było wyznaczyć, czas opóźnienia w taki sposób, żeby każde zlecenie, które jest już w kalendarzu mogło być zrealizowane. Przy tych obliczeniach bierzemy najwcześniejsze czasy wyjazdu z lokalizacji – wystarczy wziąć największy czas, który pozwala na realizację wszystkich zleceń
  - Miękkie okna – tutaj sytuacja jest nieco bardziej skomplikowana. Można stosować wiele strategii. Przykładowo można zachowywać się tak jak dla twardych okien, lub wprowadzić jakiś ograniczenie, co do max kary, związanej z opóźnieniem. Mimo wszystko tutaj również czas byłby wyznaczany na podstawie przydzielonych zleceń, których realizacja się jeszcze nie rozpoczęła.
4. Należy zmodyfikować odpowiednio algorytmy, które obliczają aktualne położenie pojazdu – algorytmy te są różne dla wersji bez grafu *algorithm.BasicSchedule* i z grafem *algorithm.GraphSchedule* – metody *updateCurrentLocation*

- BasicSchedule – należy nieznacznie zmodyfikować poniższy kawałek kodu:

```

if (z <= 1) {
    currentLocation = nextLocation;
    double dif;
    if (currentCommission.isPickup())
        dif = currentCommission.getPickupTime1() - timestamp;
    else
        dif = currentCommission.getDeliveryTime1() - timestamp;
    if (dif < 0)
        dif = 0;
    if (currentCommission.isPickup())
        waitTime = dif - (1 - z) + currentCommission.getPickUpServiceTime();
    else
        waitTime = dif - (1 - z) + currentCommission.getDeliveryServiceTime();
    waiting = true;
}

```

Należy tutaj zmodyfikować wylicznie dif, tak, żeby uwzględniony był oprócz serviceTime, czas opóźnienia wyjazdu

- GraphSchedule – tutaj sprawa się trochę komplikuje ze względu na złożoność mechanizmu aktualizacji położenia. Jednak myślę, że wystarczy lekko zmodyfikować metodę *calculateTimeToDeparture* (analogicznie jak metody zmieniane wcześniej w Schedule) i wszystko powinno działać poprawnie.

#### 4.11. Wprowadzenie kolejkowania zleceń, które są potem przekazywane przez Dystrybutora

W danym timestampie symulacji możemy mieć do rozdysponowania 0 lub więcej zleceń. Zlecenia te napływają do Dystrybutora w postaci paczek. Oznacza to, że ma on całą listę zleceń, które napłynęły w danym timestampie. Dzięki temu możemy sterować kolejnością, w jakiej te zlecenia są przydzielane jednostkom. Oczywiście najprostrzym podejściem jest dodanie odpowiednich Comparatorów i sortowanie listy zleceń. Żeby wszystko działało należałoby:

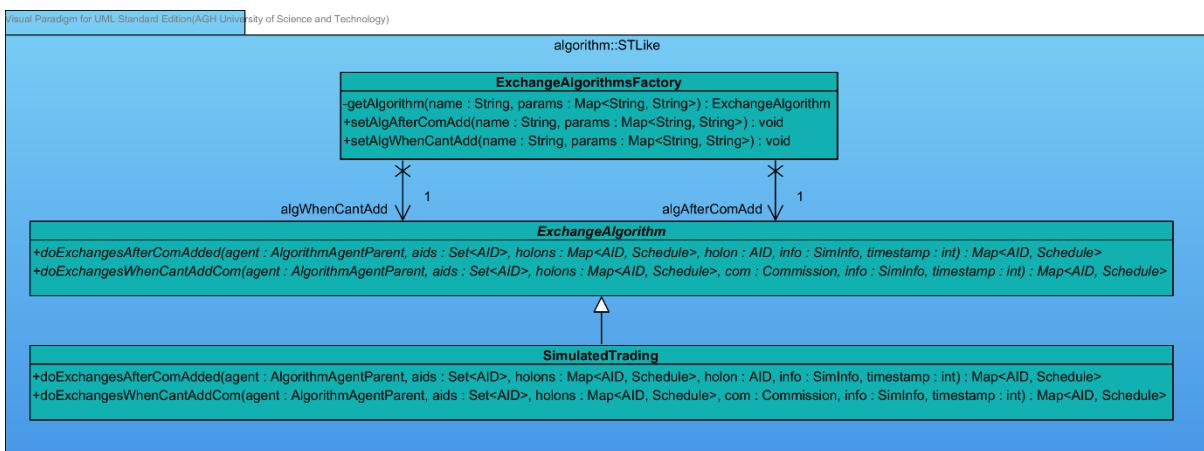
1. Implementujemy dowolne komparatory
2. Dodajemy ich użycie w metodzie `setCommissions` w `dtp.jade.distributor.DistributoAgent` w miejscu gdzie teraz jest:

```
Collections.sort(commissions, new CommissionsCompartor(simInfo.getDepot()));
```

3. W celu ułatwienia konfiguracji można w tym miejscu użyć refleksji. Przykładowo powyższe wywołanie mogłoby używać jakiegoś Factory, które zwrocałoby comparatory. Wtedy takie factory mogłoby tworzyć comparatory po nazwie klasy (tylko ją wystarczyłoby podać w konfiguracji), analogicznie jak jest to robione np. w `measure.MeasureCalculator.addCalculator`

#### 4.12. Wprowadzanie algorytmów optymalizacji działających podobnie do obecnego SimulatedTrading

System umożliwia proste dodawanie algorytmów, które działają podobnie jak ST (zaimplementowane w systemie). Sam mechanizm został zaimplementowany w taki sposób, żeby zachować kompatybilność z poprzednimi konfiguracjami.



Rysunek 47 Diagram klas przedstawiający relacje algorytmów STLike z resztą systemu

#### W celu dodania algorytmu należy:

1. dopisać klasę do pakietu `algorithm.STLike`
2. klasa musi rozszerzać `ExchangeAlgorithm` – musimy zaimplementować dwie metody:

- doExchangesAfterComAdded – metoda ta jest wołana wtedy, co poprzednio alg fullSimmulatedTrading
- doExchangesWhenCantAddCom – metoda wołana wtedy co poprzednio alg complexSimmulatedTrading

Parametry algorytmów ustawiane w konfiguracji są dostępne w zmiennej *parameters*.

### Konfiguracja:

Po dodaniu nowej klasy możemy już używać nowego algorytmu. Żeby to zrobić należy po elemencie *results* wstawić elementy:

- exchangeAlgorithmAfterComAdd
- exchangeAlgorithmWhenCantAddCom

Każdy z nich ma atrybut *name*, gdzie należy podać nazwę klasy, którą wcześniej zaimplementowaliśmy. Każdy z ww elementów może zawierać dowolną ilość elementów *param*, które mają dwa atrybuty: *name* i *value*. Dzięki temu nasze algorytmy mogą mieć różne parametry (ilość i nazwy). Jeśli któryś z elementów nie jest podany, to domyślnie konfigurowany jest algorytm SimulatedTrading

Parametrami wspólnymi dla wszystkich algorytmów jest atrybuty:

- simmulatedTrading – określający ile razy mamy uruchamiać dany algorytm
- STTimeGap – określający co ile timestampów może być uruchamiany algorytm wymian
- STCommissionGap – określający co ile zleceń może być uruchamiany algorytm wymian

W celu zachowania kompatybilności z poprzednimi konfiguracjami, można używać konfiguracji dla ST, tak jak poprzednio, lub w nowy sposób.

Przedstawione podejście pozwala też mieszać algorytmy. Przykładowo możemy stosować alg A w przypadku dodania zlecenia, natomiast alg B, gdy nie można przydzielić zlecenia dla żadnego holonu. Daje to dużą elastyczność.

#### 4.12.1. Implementacja klasycznego Simulated Trading

Samo dodanie takiego algorytmu jest bardzo proste. Wystarczy dodać ten algorytm, jako algorytm STLike. Pewnym problemem, który się pojawia, jest założenie odnośnie fazy uruchamianej, gdy nie istnieje jednostka, która może obsłużyć nowe zlecenie (exchangeAlgorithmWhenCantAddCom). Ten algorytm jest wyzwalany właśnie tym zleceniem. Klasyczny algorytm ST nie przewiduje takiego zachowania. Jednak problem ten można rozwiązać inicjując algorytm nowym zleceniem. Chodzi o to, że można przyjąć, że nowe zlecenie zostało już wrzucone do algorytmu przez jakiegoś agenta (wirtualnego) i wtedy normalnie kontynuować algorytm.

#### 4.13. Reorganizacja holonów - wytyczne

Na wstępnie należy zaznaczyć, że samo zagadnienie jest dość ambitne do realizacji. Sam problem jest dość złożony. Należy najpierw rozpatrzyć warunki uruchomienia reorganizacji, czyli kiedy ma ona tak naprawdę być uruchamiana. Następnie należy zaimplementować jakiś algorytm. Podejścia są co najmniej dwa:

1. Podejście oparte o algorytm negocjacyjny, który jest obecnie używany do tworzenia holonów – tutaj należałoby najpierw dopisać „watchera”, który zapobiegałby zakleszczeniom (wystraczy co jakiś czas odpytywać agentów o ich stan i w wykrywać zakleszczenie – jeśli takie wystąpi to sztucznie zmodyfikować tablicę preferencji któregoś z agentów i wznowić negocjację). Następnie można próbować odłączać jakiś element holona i prowadzić negocjację mającą na celu dołączenie innego elementu. Wtedy praktycznie nie ruszamy algorytmu negocjacyjnego. W tym przypadku największe modyfikacje byłby w miejscu tworzenia listy preferencji poszczególnych pojazdów. Oprócz tego należałoby uwzględnić możliwość wymiany elementów między już istniejącymi holonami.
2. Podejście oparte o własny algorytm wymiany elementów – to podejście wydaje się być prostrze. Można stworzyć jakiegoś agenta realizującego reorganizację. Taki agent działałby w oparciu o jakąś politykę wymian i sam dokonywałby wyborów jaki

element wymienić i jaki zamiennik wybrać. Wymagałoby to jedynie przesłania kilku komunikatów i podmiany w ExecutionUnitAgent odpowiednich części – to nie stanowi żadnego problemu, w sensie nie wpłynie na działanie algorytmu jakoś znacząco. To podejście jest dużo bardziej elastyczne i uważam, że pod wieloma względami lepsze, szczególnie, jeśli nie zna się dobrze systemu. Daje też większe możliwości, co do reorganizacji, gdyż pozwala na stosowanie nawet bardzo złożonych polityk reorganizacji.

Sama reorganizacja może przebiegać dwójako. Albo np. Dispatcher może zadecydować, że teraz robimy reorganizacje, albo same jednostki mogą o tym decydować na podstawie jakiś kryteriów. Można również rozważyć scenariusz, gdzie po każdym kroku odwołujemy się do agenta odpowiedzialnego za reorganizacje (podejście 2) i on decyduje na podstawie jakiejś polityki, czy robić reorganizacje. Może on też zawsze robić reorganizacje, z możliwością wymiany na ten sam element i prowadzić optymalizację w każdym kroku algorytmu.

Biorąc pod uwagę powyższe przemyślenia, można sobie wyobrazić kilka schematów realizacji reorganizacji:

- Reorganizacja inicjowana przez jednostki – w tym przypadku w miejscu gdzie jest przesyłane nowe zlecenie (metoda *action* w *dtp.jade.eunit.GetCommissionForEUnitBehaviour*) można realizować jakąś politykę i sprawdzać, czy opłaca nam się przeprowadzić reorganizację
- Reorganizacja inicjowana przez element centralny (osobny agent lub dystrybutor) – w tym przypadku możemy dodać do przetwarzania fazę, gdzie wysyłamy dane do holonów w jedno miejsce i tam podejmowana jest decyzja o przeprowadzeniu reorganizacji.
- Podejście mieszane – o samej reorganizacji może decydować holon, natomiast za jej przebieg może już odpowiadać element centralny

#### 4.14. Mechanizm dodawania nowych algorytmów uczenia maszynowego

Dodanie nowych algorytmów uczenia maszynowego jest stosunkowo proste. Jedyne co musimy zrobić, to napisać klasę, która:

- będzie dziedziczyć po machineLearning.MLAlgorithm
- będzie znajdować się w pakiecie machineLearning.<nazwa klasy małymi literami>

Po spełnieniu powyższych, możemy już wykorzystywać algorytm w konfiguracji testu.

## 4.15. Dodanie dedykowanego algorytmu dla miękkich okien

W celu stworzenia kompleksowego rozwiązania dla miękkich okien należy zaimplementować dedykowany algorytm, z uwagi na to, że modyfikacja istniejącego rozwiązania może okazać się trudniejsza i na pewno spowoduje, że kod będzie trudny w utrzymaniu i nieczytelny.

Sama implementacja algorytmu musi zostać podzielona na dwa etapy. Najpierw należy zaimplementować nowy algorytm przydzielania zleceń, a następnie odpowiedni algorytm STLike. Z uwagi na to, że obecnie większość danych (jak np. czas, dystans) jest wyliczana w klasie Schedule, takie podejście nie jest akceptowalne (z powodu podobnych problemów, o których wspominałem wyżej). Wydaje się, że najlepszym rozwiązaniem byłaby implementacja wyliczania wspomnianych wartości albo bezpośrednio w klasach implementowanych algorytmów – wtedy Schedule traktujemy jedynie jako kontener ze zleceniami. Konkurencyjnym rozwiązaniem jest również stworzenie nowej implementacji Schedule.

### 4.15.1. Wersja z implementacją wyliczania danych w klasach algorytmów

To podejście ma pewną wadę. Otóż wartości pobierane z klasy Schedule są wykorzystywane do przesyłania danych wykorzystywanych do tworzenia podsumowania z symulacji. Nie jest to jednak bardzo dużym problemem, wystarczy dodać ifa ze sprawdzeniem klasy wykorzystywanych algorytmów w metodach:

- sendCalendarStats
- sendCalendarStatsToFile
- sendWorstCommission
- checkNewCommission

#### 4.15.2. Wersja z implementacją nowej klasy Schedule

Zasadniczo wystarczyłoby dodać odpowiednie ustawienia w konfiguracji (wybór, które Schedule używamy), lub arbitralnie włączać dane Schedule na podstawie wybranego algorytmu szeregowania zleceń. Żeby dana implementacja się rozpropagowała w systemie wystarczy ustawić scheduleCreator w klasie SimInfo (analogicznie jak to jest robione z GraphSchedule).