raport 2 - 3

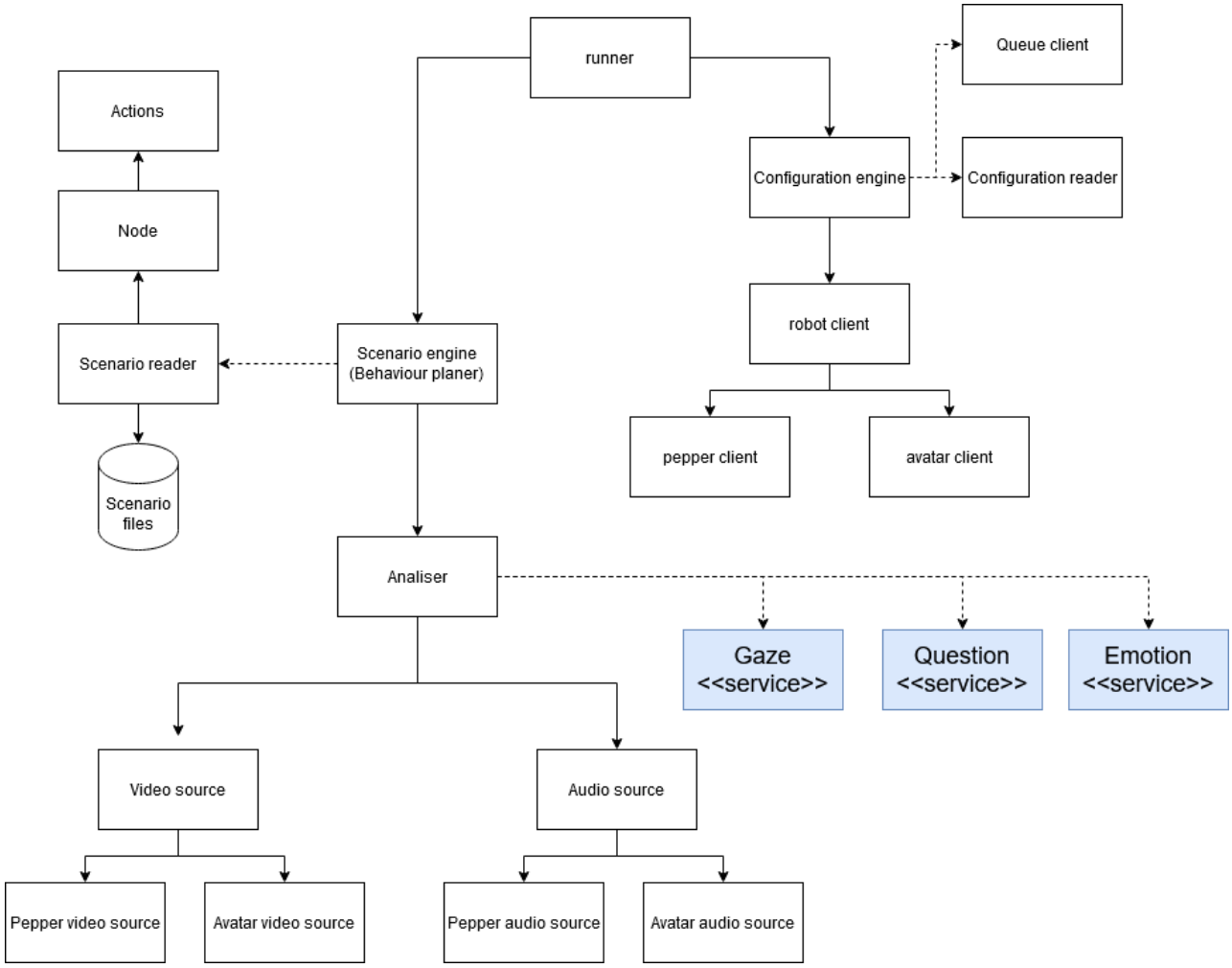
majarosz.j90

December 2020

1 Architektura systemu

Schemat systemu przedstawiono na rysunku 1. Ze względu na wymagania funkcjonalne dotyczące działania systemu zarówno na robocie jak i avatarze wybrane komponenty mają dwie implementacje: Video Source, Audio Source, robot client. Głównymi komponentami są:

- Video Source: Moduł odpowiedzialny za dostarczenie jednolitego interfejsu do obsługi wideo dla systemu działającego na robocie i na androidzie (avatar).
- Pepper video Source: implementacja modułu Video Source odpowiedzialna za pobieranie wideo z kamery robota.
- Avatar video Source: implementacja modułu Video Source odpowiedzialna za pobieranie wideo z kamery znajdującej się na urządzeniu android gdzie działa avatar.
- Audio Source: Moduł odpowiedzialny za dostarczenie jednolitego interfejsu do obsługi audio dla systemu działającego na robocie i na androidzie (avatar).
- Pepper audio Source: implementacja modułu Audio Source odpowiedzialna za pobieranie audio z mikrofonu robota.
- Avatar audio Source: implementacja modułu Audio Source odpowiedzialna za pobieranie audio z mikrofonu znajdującego się na urządzeniu android gdzie działa avatar.



Rysunek 1: Schemat architektury systemu

- Gaze«service»: Moduł odpowiedzialny za dostarczenie jednolitego interfejsu do wykrywania kierunku spojrzenia i skierowania głowy dla systemu działającego na robocie i na urządzeniu mobilnym z systemem Android. Wywoływany jest przez moduł Analyser, przyjmuje obraz z kamery robota lub urządzenia mobilnego i zwraca kierunek spojrzenia. Moduł działa asynchronicznie i może przeprowadzić obliczenia lokalnie albo przesłać dane do zdalnego serwera w celu wykonania obliczeń.
- Question«service»: Moduł odpowiedzialny za dostarczenie jednolitego interfejsu do wykrywania pytań dla systemu działającego na robocie i na urządzeniu mobilnym z systemem Android. Wywoływany jest przez moduł Analyser, przyjmuje dźwięk z mikrofonu robota lub urządzenia mobilnego i zwraca informację czy wykryto pytanie. Moduł działa asynchronicznie i może przeprowadzić obliczenia lokalnie albo przesłać dane do zdalnego serwera w celu

wykonania obliczeń.

- Emotion«service»: Moduł odpowiedzialny za dostarczenie jednolitego interfejsu do wykrywania emocji dla systemu działającego na robocie i androidzie. Wywoływany jest przez moduł Analizera, przyjmując obraz z kamery robota lub urządzenia mobilnego i zwraca wykrytą emocję. Moduł może przeprowadzić obliczenia lokalnie albo przesłać dane do zdalnego serwera w celu wykonania obliczeń, moduł działa asynchronicznie.

- Analyser: moduł wykorzystuje moduły Gaze, Question, Emotion w celu otrzymania danych na temat kierunku spojrzenie, wykrytych emocji oraz wykrycia pytania w wypowiedzi, następnie przeprowadza na otrzymanych danych analizę, uruchamia zarejestrowane eventy przez moduł scenario engine na bazie wyników analizy.

- Scenario Engine: moduł odpowiedzialny jest za kontrolę przebiegu scenariuszy. Wykorzystuje on moduł Scenario reader do wczytania scenariusza w postaci grafu skierowanego. Graf składa się z węzłów będącymi elementami typu Node, moduł scenario engine zaczyna wykonywanie scenariusza od korzenia i porusza się po drzewie zgodnie z spełnionymi warunkami w węzłach aż dojdzie do Węzła kończącego (liścia). Węzeł początkowy przeważnie zawiera moduły potrzebne do wykonania scenariusza, a węzeł kończący elementy systemu wymagające specjalnego traktowania podczas zamykania.

- Scenario reader: moduł odpowiedzialny jest za wczytanie scenariusza zapisanego w pliku w formacie json i przetworzenie go w graf skierowany składający się z węzłów z modułu Node.

- Node: moduł definiujący różne typy węzła mogące się składać na graf scenariusza, np. prosty, równoległy, zapętlony, itd. Każdy rodzaj węzła zawiera listę akcji(Actions) do wykonania, warunek rozpoczęcia, warunek zakończenia, oraz listę eventów. Event składa się z warunku wywołania, czasu w jakim może się powtórzyć i listy akcji do wykonania w trakcie wywołania eventu.

- Actions: moduł zawiera wszystkie możliwe akcje do wywołana w systemie. Np. ruchy robota, powiedzenie czegoś, uruchomienie analizy wideo, zatrzymanie śledzenia twarzy itd.

- Configuration Engine: moduł odpowiedzialny za przygotowanie systemu do działania w zależności od konfiguracji odczytanej z pliku przez Configuration reader. Konfiguracja różni się dla robota i avatara, w konfiguracji definiujemy adres kolejki do której podłączony jest robot lub avatar, moduły niezbędne do działania systemu w danej konfiguracji, informację o konfiguracji dla systemu.

- Configuracion reader: moduł odpowiada za odczytanie konfiguracji z pliku w formacie json. Jest wykorzystywany przez Configuration Engine.

- Queue client: moduł odpowiedzialny za obsługę komunikacji z aplikacją działającą na robocie lub aplikacją avatara.

- robot client: Moduł wykonawczy dla akcji(actions) wykonywanych na robocie lub avatarze. wykorzystuje Queue client do komunikacji i/lub natywny interfejs robota.

- Pepper client: implementacja robot client dla robota pepper.
- Avatar client: implementacja robot client dla aplikacji avatara na urządzeniu mobilnym.
- runner: moduł będący punktem startowym systemu zawiera URI do pliku z konfiguracją i scenariuszem. Odpowiada za wywołanie modułu Configuration Engine i przekazanie jego efektów działania do modułu Scenario engine oraz uruchomienie tego modułu.

2 Implementacja systemu

W tym rozdziale przedstawiam wykorzystane narzędzia, ich charakterystykę oraz najważniejsze funkcje. Szczegółowo omawiam implementację i działanie systemu.

2.1 Wykorzystywane narzędzia i ich najważniejsze funkcjonalności

W tym podrozdziale przedstawiam narzędzia dzięki którym możliwe jest funkcjonowanie systemu oraz porównanie ich najistotniejszych funkcjonalności:

- OpenCV - jest to najpopularniejsza biblioteka, wykorzystywana w dziedzinie widzenia maszynowego. Napisana została w języku C/C++ [1]. Posiada ona także interfejs w języku Python. Możliwe jest również wsparcie akceleracji sprzętowej i pełne wykorzystanie architektury wielordzeniowych procesorów. Działa na wielu platformach: Windows, Linux, Android oraz Mac OS. Używana jest przez duże korporacje, takie jak: Google, Yahoo, Microsoft, Intel, IBM. Zapewnia wiele gotowych funkcji wymaganych do implementacji projektu:
  - metody pozwalające na obsługę kamery, pobieranie kolejnych klatek obrazu, wybranie źródła video itp,
  - *CascadeClassifier* i *detectMultiScale* pozwalające używać kaskad Haar'a w celu identyfikacji twarzy czy obszaru oczu, są to realizacje algorytmów opisanych szerzej w rozdziale **??**,
  - *solvepnp* używaną do rzutowania punktów z 2D do 3D podczas procesu określania skierowania głowy,
  - w najnowszych wersjach został zaimplementowany moduł odpowiedzialny za wykrywanie punktów charakterystycznych na twarzy. Jednak jest on nie dostępny na robocie ze względu na jego oprogramowanie.
- Dlib - popularna biblioteka stosowana w przetwarzaniu obrazu oraz uczeniu maszynowym[2]. Oferuje ona częściowo bardzo podobny zestaw funkcjonalności jak OpenCV. Jednak funkcje pozwalające wykryć twarz w porównaniu do tych z OpenCV działają nieznacznie szybciej i bardziej niezawodnie. Biblioteka zaś nie wchodzi w konflikt z oprogramowaniem robota. Napisana w języku C++, posiada interfejs w języku Python. Używana zarówno w przemyśle jak i środowisku akademickim. Działa na wielu platformach Windows, Linux, Android oraz

[1] https://opencv.org/
[2] http://dlib.net/

Mac OS. Zawiera wiele algorytmów z dziedziny uczenia maszynowego, na przykład sieci neuronowe, wiele różnych rodzajów SVM. Zawiera także algorytmy numeryczne z algebry liniowej. Metody z dziedziny widzenia maszynowego:

  - Get_frontal_face_detector tworzy detektor wykrywający twarz z przodu, wykorzystuje on kaskadę haara
  - Shape_predictor pozwala on wraz z odpowiednim plikiem, na wykrywanie punktów charakterystycznych na twarzy.

- Deepgaze - jest to framework wykorzystujący konwolucyjne sieci neuronowe w celu określenia kierunku skierowania głowy[3]. Napisany został w języku Python.
- OpenFace - jest to framework pozwalający określić kierunek spojrzenia i skierowania głowy, wykorzystuje on biblioteki Dlib i openCV do swojego działania[4]. Wykorzystywany jest także autorski model zaproponowany przez autorów frameworku, podobny do tego z biblioteki Dlib, jednak posiadający także współrzędne w 3D.
- NaoQi - jest to framework dostarczany przez producenta robota[5]. Pozwala on na kontrolę nad ruchami robota. Zawiera moduły odpowiedzialne za syntezowanie mowy w kilku językach oraz rozpoznawanie czy wypowiedź zawiera podane słowa kluczowe. Pozwala na określenie kierunku spojrzenia, kierunku skierowania głowy, wykrywanie twarzy oraz takich rzeczy jak mruganie czy uśmiech. Napisany został w języku C++ ale posiada interfejs w Pythonie.
- nltk - Natural Language Toolkit biblioteka wykorzystywana do przetwarzania języka naturalnego[6]. Napisana została w języku Python, udostępnia wiele funkcji jak: tokenizowanie, parsowanie, klasyfikacje, tagowanie. Działa na wielu platformach Windows, Linux, Android oraz Mac OS.

2.2 Szczegółowy opis rozwiązania

W tym rozdziale na podstawie diagramów klas i sekwencji przedstawiona zostanie szczegółowo implementację systemu.

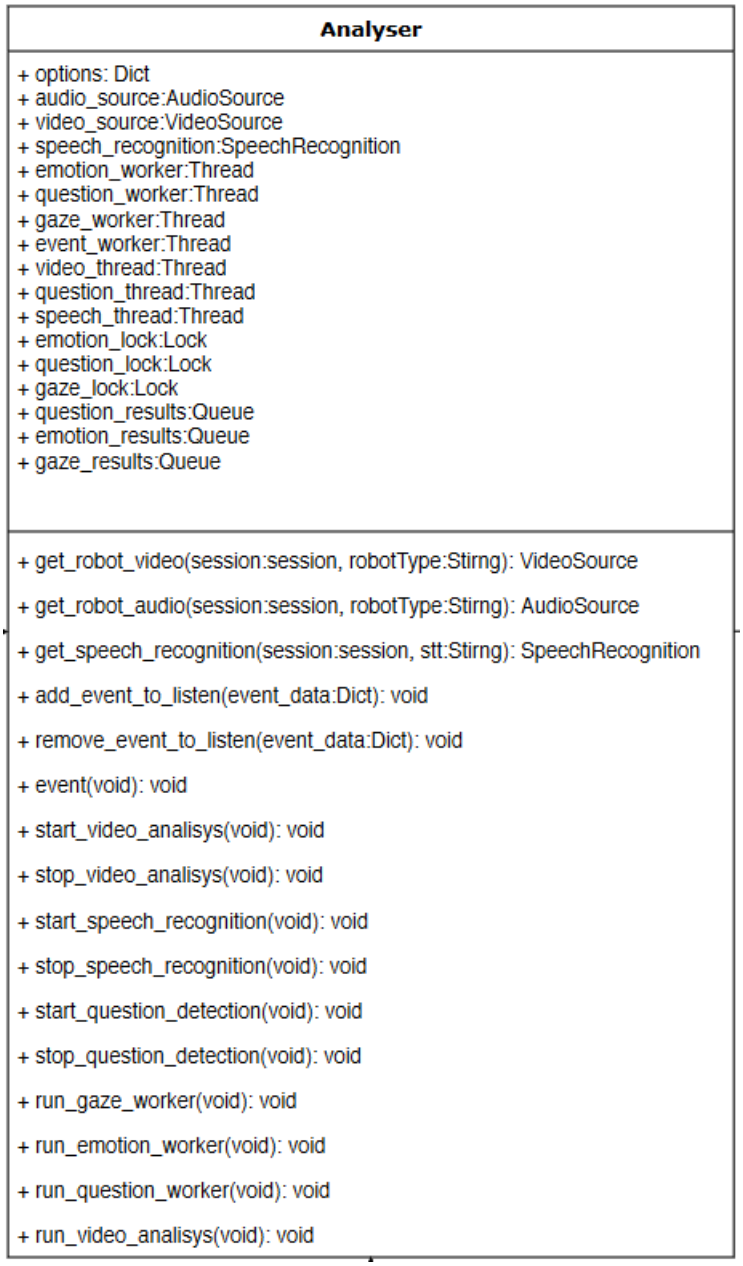
2.2.1 Diagram klas

Na rysunku 2 znajduje się diagram klas, który przedstawia szczegółową architekturę głównego modułu systemu.

[3] https://github.com/mpatacchiola/deepgaze
[4] https://github.com/TadasBaltrusaitis/OpenFace
[5] http://doc.aldebaran.com/2-5/dev/programming_index.html
[6] https://www.nltk.org/

Rysunek 2: Diagram klas

Klasa *Analyser* - Jest to klasa odpowiedzialna za przetwarzanie danych pochodzących z klas *VideoSource* i *VideoSource* z użyciem zewnętrznych serwisów żeby uzyskać dane o emocjach, kierunku spojrzenia użytkownika oraz czy nie zadaje pytania. Tak przygotowane dane są poddawane analizie i na zgodnie z założeniami scenariusza podejmowane są decyzję. Diagram przedstawiono na rysunku 3, a szczegółowy opis w tabeli 1.



Rysunek 3: Diagram klasy *Analyser*

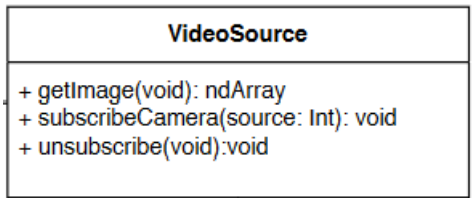
Tabela 1: Przegląd metod klasy *Analyser*

Metoda	Opis

get_robot_video(session: session, robotType:Stirng): VideoSource	metoda tworzy klasę VideoSource w zależności od typu robota zdefiniowanego w pliku konfiguracyjnym.
get_robot_audio(session: session, robotType:Stirng): AudioSource	metoda tworzy klasę AudioSource w zależności od typu robota zdefiniowanego w pliku konfiguracyjnym.
get_speech_recognition(session: session, stt:Stirng): SpeechRecognition	Metoda odpowiedzialna za przygotowanie klasy *SpeechRecognition* odpowiadającej za transkrypcje mowy na tekst
add_event_to_listen(event_data: Dict): void	Metoda dodaje Eventy zdefiniowane w klasie *Node* do listy eventów na które należy zareagować. Parametr $event_data$ zawiera typ eventu oraz dodatkowe informacje jak czas miedzy wywołaniami eventu. Metoda jest wywoływana przed rozpoczęciem działania *Node*
remove_event_to_listen(event_data: Dict): void	Metoda odpowiedzialna za usunięcie elementów z listy eventów do zareagowania. Parametr $event_data$ jednoznacznie określa konkrety event do usunięcia. Metoda jest wywoływana po zakończeniu działania *Node*
event(void): void	Jest to metoda odpowiedzialna za sprawdzanie czy nie zostały spełnione warunki do wywołania eventu. Działa ona w osobnym wątku, sprawdzanie następuje co 0.1 sekundy.
start_video_analisys(void): void	Metoda odpowiedzialna jest za uruchomienie wątków odpowiedzialnych za odbieranie danych wideo i wysyłanie ich do ekstrakcji emocji i kierunku spojrzenia. Uruchamiane są także wątki odpowiedzialne za odbieranie przetworzonych już danych o emocjach i kierunku spojrzenia.
stop_video_analisys(void): void	Metoda odpowiedzialna jest za zatrzymanie wszystkich wątków uruchomionych przez metodę $start_video_analisys$.
start_speech_recognition(void): void	Metoda odpowiedzialna jest za uruchomienie wątków odpowiedzialnych za odbieranie danych audio i wysyłanie ich do przetwarzania mowy na text.
stop_speech_recognition(void): void	Metoda odpowiedzialna jest za zatrzymanie wszystkich wątków uruchomionych przez metodę $start_speech_recognition$.
start_question_detection(void): void	Metoda odpowiedzialna jest za uruchomienie wątków odpowiedzialnych za odbieranie danych audio i wysyłanie ich do rozpoznawania czy użytkownik zadał pytanie. Uruchamiane są także wątki odpowiedzialne za odbieranie przetworzonych już danych.
run_gaze_worker(void): void	Metoda uruchamiana jest w osobnym wątku przez $start_video_analisys$ odpowiedzialna jest za odbieranie przetworzonych już danych o kierunku spojrzenia. Dane są zapisywane i przekazywane dalej do analizy.

run_emotion_worker(void): void	Metoda uruchamiana jest w osobnym wątku przez $start_videoanalisys$ odpowiedzialna jest za odbieranie przetworzonych już danych o emocjach użytkownika. Dane są zapisywane i przekazywane dalej do analizy.
run_question_worker(void): void	Metoda uruchamiana jest w osobnym wątku przez $start_questiondetection$ odpowiedzialna jest za odbieranie przetworzonych już danych o tym czy użytkownik zadał ostatnio pytanie. Dane są zapisywane i przekazywane dalej do analizy.
run_video_analisys(void): void	Metoda uruchamiana jest w osobnym wątku przez $start_videoanalisys$ odpowiedzialna jest za odbieranie nieprzetworzonych danych video z *VideoSource*. Dane są wysyłane asynchronicznie do serwisów odpowiedzialnych za wykrywanie emocji i kierunku spojrzenia.

Klasa abstrakcyjna *VideoSource* - Jest to klasa odpowiedzialny za transparentny dostęp do zdjęć pochodzących z kamery zarówno robota jak i kamerki urządzenia mobilnego na którym działa aplikacja z avatarem. Diagram przedstawiono na rysunku 4, a szczegółowy opis w tabeli 2.

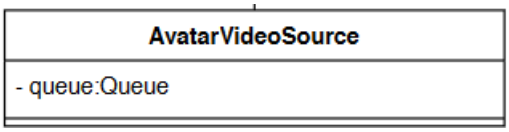


Rysunek 4: Diagram klasy abstrakcyjnej *VideoSource*

Tabela 2: Przegląd metod klasy abstrakcyjnej *VideoSource*

Metoda	Opis
getImage(void):ndArray	Metoda zawraca ostatnią zarejestrowaną klatkę w postaci tablicy ndarray.
subscribeCamera(saource: Int):void	Metoda służy do zarejestrowania się na otrzymywanie klatek z kamery. Parametr *source* określa źródło obrazu.
unsubscribe(void):void	Metoda służy do bezpiecznego zamknięcia strumienia danych pochodzących z kamery.

Klasa *AvatarVideoSource* - Jest to implementacja klasy abstrakcyjnej *VideoSource* obsługująca kamerę urządzenia mobilnego, dane są przesyłane w formacie mjpeg, z użyciem kolejki mqtt. Diagram przedstawiono na rysunku 5, a szczegółowy opis w tabeli 3.

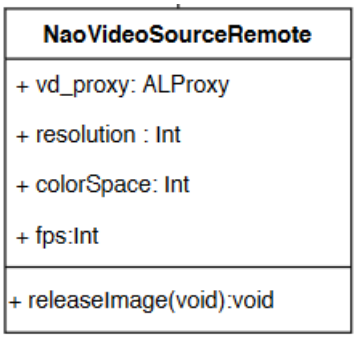


Rysunek 5: Diagram klasy *AvatarVideoSource*

Tabela 3: Przegląd metod klasy *AvatarVideoSource*

Metoda	Opis
getImage(void):ndArray	Metoda zwraca ostatnią zarejestrowaną klatkę z kamery sieciowej w postaci tablicy ndArray.
subscribeCamera(saource: Int):void	Metoda służy do zarejestrowania się na otrzymywanie klatek z kamery. Parametr *source* określa źródło obrazu.
unsubscribe(void):void	Metoda służy do bezpiecznego zamknięcia strumienia danych pochodzących z kamery.

Klasa *NaoVideoSourceRemote* - Jest to implementacja klasy abstrakcyjnej *VideoSource* obsługująca zdalnie kamerę robota. Diagram przedstawiono na rysunku 6, a szczegółowy opis w tabeli 4.

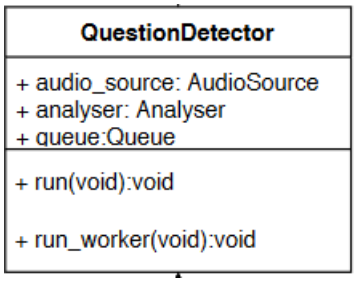


Rysunek 6: Diagram klasy *NaoVideoSourceRemote*

Tabela 4: Przegląd metod klasy *NaoVideoSourceRemote*

Metoda	Opis
getImage(void):ndArray	Metoda zwraca ostatnią zarejestrowaną klatkę z kamery robota w postaci tablicy ndArray.
subscribeCamera(saource :Int):void	Metoda służy do zarejestrowania się na otrzymywanie klatek z kamery. Parametr *source* określa źródło obrazu.
unsubscribe(void):void	Metoda służy do bezpiecznego zamknięcia strumienia danych pochodzących z kamery.
realeseImage(void):void	Metoda zwolnienia klatki w oprogramowaniu robota.

Klasa *QuestionDetector* - Jest to klasa odpowiedzialna za wykrywanie pytań w mowie użytkownika korzysta z *AudioSource* w celu pobierania danych audio i zewnętrznych serwisów w celu analizy danych. Diagram przedstawiono na rysunku 7, a szczegółowy opis w tabeli 5.

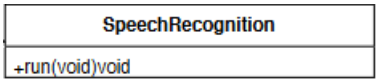


Rysunek 7: Diagram klasy *QuestionDetector*

Tabela 5: Przegląd metod klasy *QuestionDetector*

Metoda	Opis
run(void):void	Metoda uruchamiana w osobnym wątku, pobiera dane audio a użyciem *AudioSource* i wykrywa czy użytkownik coś mówi jeżeli tak przesyła je do do metody *run_worker* jeżeli nie sygnalizuje wykrycie ciszy.
run_worker(void):void	Metoda uruchamiana w osobnym wątku, odbiera dane od metody *run* i przesyła je do zewnętrznego serwisu lub przetwarza lokalnie w celu wykrycia czy użytkownik zadał pytanie.

klasa *SpeechRecognition* - Jest to klasa odpowiedzialna za pozyskanie informacji o wypowiedzianych słowach, używa w tym celu narzędzi chmurowych dostarczanych przez Google. Diagram przedstawiono na rysunku 8, a szczegółowy opis w tabeli 6.

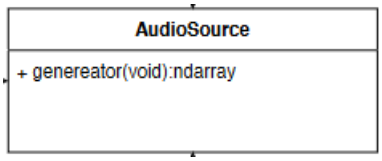


Rysunek 8: Diagram klasy *SpeechRecognition*

Tabela 6: Przegląd metod klasy *SpeechRecognition*

Metoda	Opis
run(void):void	Metoda ta pozyskuje dane z użyciem klasy *AudioSource* i przesyła je do usługi Google SpeechToText. Otrzymana transkrypcja jest zapisywana i przesyłana dalej do analizy.

Klasa abstrakcyjna *AudioSource* - Jest to klasa odpowiedzialny za transparentny dostęp do danych audio pochodzących z mikrofonu zarówno robota jak i mikrofonu urządzenia mobilnego na którym działa aplikacja z avatarem. Klasa implementuje pattern generator. Diagram przedstawiono na rysunku 9, a szczegółowy opis w tabeli 7.

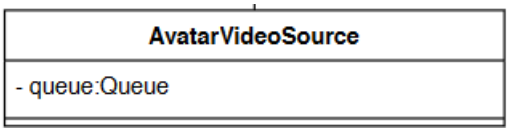


Rysunek 9: Diagram klasy abstrakcyjnej *AudioSource*

Tabela 7: Przegląd metod klasy abstrakcyjnej *AudioSource*

Metoda	Opis
generator(void):ndArray	Metoda implementuje pattern generator dane są zwracane w postaci ndArray.

Klasa *AvatarAudioSource* - Jest to implementacja klasy abstrakcyjnej *AudioSource* obsługująca mikrofon urządzenia mobilnego, dane są przesyłane w skompresowane algorytmem gzip, z użyciem kolejki mqtt. Diagram przedstawiono na rysunku 10, a szczegółowy opis w tabeli 8.

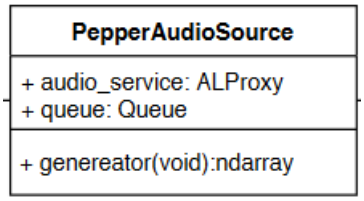


Rysunek 10: Diagram klasy *AvatarAudioSource*

Tabela 8: Przegląd metod klasy *AvatarAudioSource*

Metoda	Opis
generator(void):ndArray	Metoda implementuje pattern generator dane są zwracane w postaci ndArray.

Klasa *PepperAudioSource* - Jest to implementacja klasy abstrakcyjnej *AudioSource* obsługująca zdalnie mikrofon robota. Diagram przedstawiono na rysunku 11, a szczegółowy opis w tabeli 9.

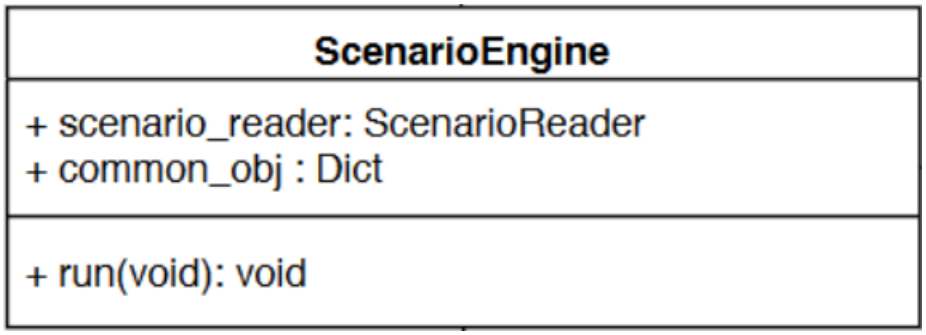


Rysunek 11: Diagram klasy *PepperAudioSource*

Tabela 9: Przegląd metod klasy *PepperAudioSource*

Metoda	Opis
generator(void):ndArray	Metoda implementuje pattern generator dane są zwracane w postaci ndArray.

Klasa *ScenarioControler* - Jest to klasa odpowiedzialna za realizację scenariusza pod postacią klasy *Scenario*, wczytanego z pliku przez klasę *ScenarioReader*. Uruchamia kolejne węzły *Node* scenariusza sprawdzając warunki startu. Diagram przedstawiono na rysunku 12, a szczegółowy opis w tabeli 10.

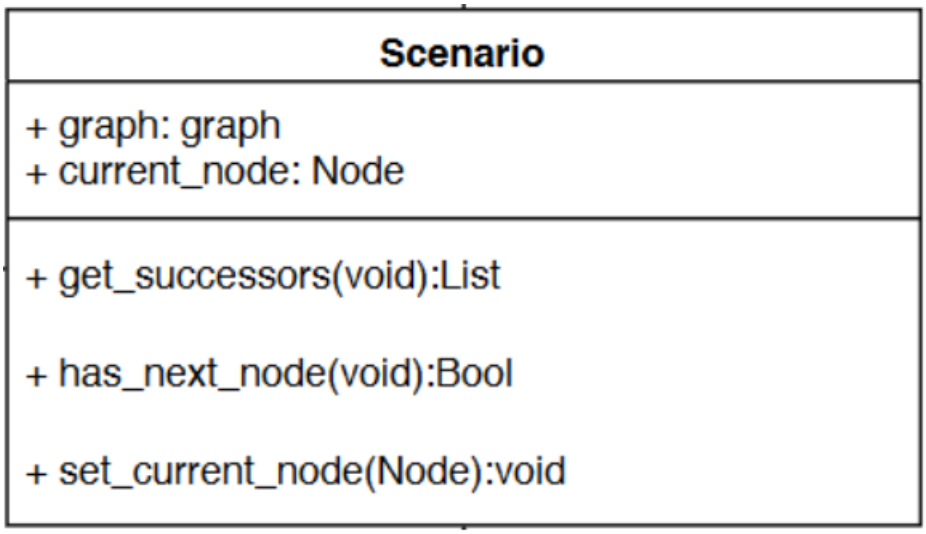


Rysunek 12: Diagram klasy *ScenarioControler*

Tabela 10: Przegląd metod klasy *ScenarioControler*

Metoda	Opis
run(void):void	Metoda odpowiedzialna za uruchomienie i prawidłowe przeprowadzenie wczytanego scenariusza. Uruchamia kolejne węzły *Node* odpowiednio do ich typu oraz w zależności czy został spełniony warunek startu

Scenario - Jest to Klasa odpowiedzialna za obsługę i przechowywanie wczytanego przez *ScenarioReader* scenariusza. Scenariusz jest przechowywany jako graf skierowany. Diagram przedstawiono na rysunku 13, a szczegółowy opis w tabeli 11.

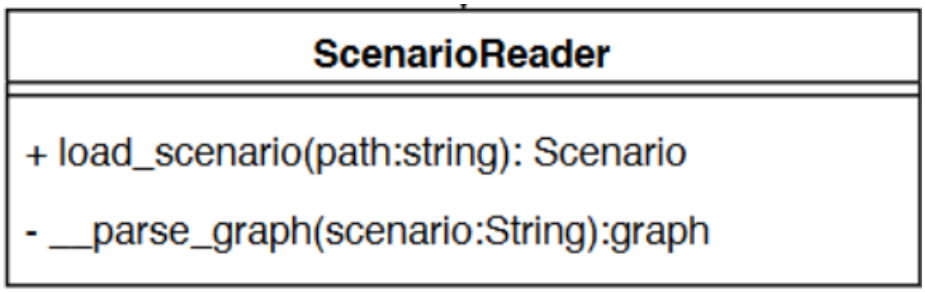


Rysunek 13: Diagram klasy *Scenario*

Tabela 11: Przegląd metod klasy *Scenario*

Metoda	Opis
get_successors(void) :List	Metoda zwraca listę następców obecnie przetwarzanego węzła *curent_node*.
has_next_node(void):Bool	Metoda zwraca prawdę jeżeli obecnie przetwarzany węzeł *curent_node* ma następcąów w przeciwnym wypadku zwracany jest fałsz.
set_current_node(num: Int):void	Metoda ustawia zmienną *curent_node* na węzeł na pozycji *num*.

ScenarioReader - Jest to Klasa odpowiedzialna za wczytanie z pliku w formacie json scenariusza i przetworzenie go do postaci grafu skierowanego. Scenariusz składa się z elementów klasy *Node*. Diagram przedstawiono na rysunku 14, a szczegółowy opis w tabeli 12.

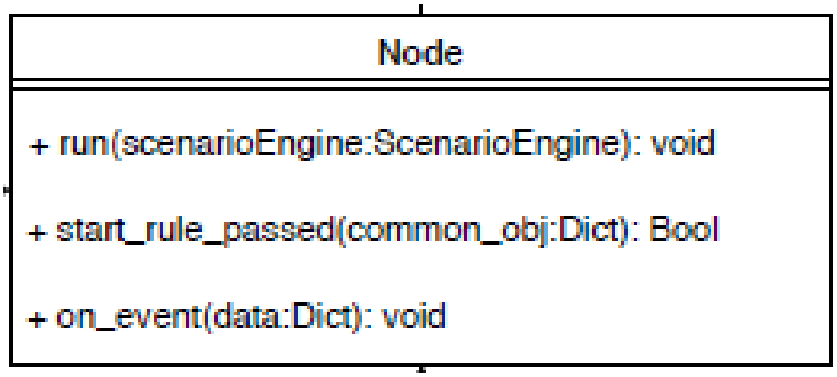


Rysunek 14: Diagram klasy *ScenarioReader*

Tabela 12: Przegląd metod klasy *ScenarioReader*

Metoda	Opis
load_scenario(void) :Scenario	Metoda zwraca obiekt klasy *Scenario* z wczytanym z pliku w formacie json scenariuszem.
__parse_graph(scenario: String):graph	Metoda odpowiedzialna jest za przetworzenie wczytanego pliku w formacie json na graf skierowany złożony z węzeł klasy *Node*

Node - Jest to klasa abstrakcyjna reprezentująca jeden etap scenariusza. Diagram przedstawiono na rysunku 15, a szczegółowy opis w tabeli 13.

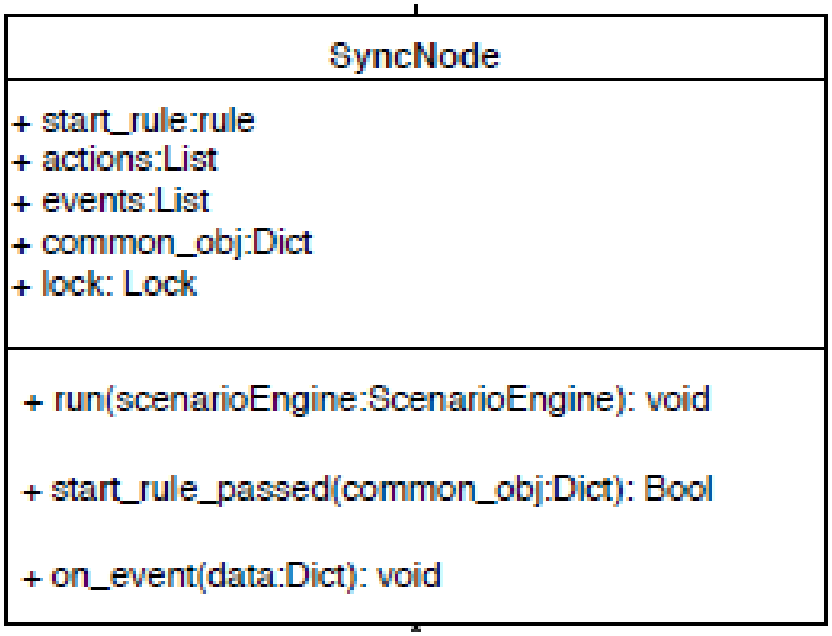


Rysunek 15: Diagram klasa abstrakcyjna *Node*

Tabela 13: Przegląd metod klasa abstrakcyjna *Node*

Metoda	Opis
run(scenarioEngine: ScenarioEngine): void	Metoda odpowiedzialna za uruchomienie wszystkich zaplanowanych w tym etapem scenariusza akcji. Zmienna *scenarioEngine* jest przekazywana akcją w razie potrzeby
start_rule_passed(common_obj:Dict): Bool	Metoda odpowiedzialna jest za sprawdzenie czy warunek start_rule zdefiniowany w scenariuszy dla danego węzła jest spełniony dane zawarte w *common_obj* mogą być niezbędne w tym celu
on_event(data:Dict): void	Metoda skurzy wywołaniu metody *run* klasy *event* dla eventu określanego przez zmienną *data*. wywoływana jest przez klasę *Analyser* po spełnieniu warunków określonych w scenariuszu dla tego Eventu

SyncNode - Jest to klasa reprezentująca jeden etap scenariusza. Jest ona najprostszą implementacją klasy abstrakcyjnej *Node*, zakłada liniowe wykonanie wszystkich akcji zdefiniowanych w liście *actions*. Eventy mogą być wywoływane pomiędzy akcjami lub w trakcie niektórych np *WaitAction*, zmienna *lock* jest używana w celu synchronizacji podczas wywoływania eventu. Diagram przedstawiono na rysunku 16, a szczegółowy opis w tabeli 14.

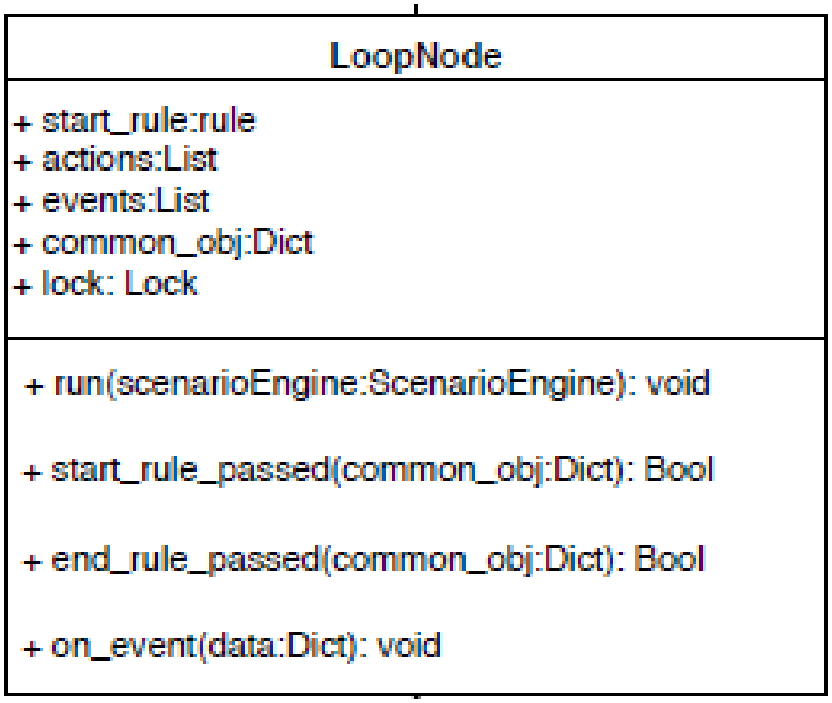


Rysunek 16: Diagram klasy *SyncNode*

Tabela 14: Przegląd metod klasy *SyncNode*

Metoda	Opis
run(scenarioEngine: ScenarioEngine): void	Metoda odpowiedzialna za uruchomienie wszystkich zaplanowanych w tym etapie scenariusza akcji. Na Początku rejestruje wszystkie eventy z listu *events* w klasie *Analuser* do sprawdzania czy nie zostały spełnione warunki ich wywołania, Po wykonaniu wszystkich akcji eventy są wyrejestrowywane. Zmienna *scenarioEngine* jest przekazywana akcją w razie potrzeby
start_rule_passed(common_obj: Dict): Bool	Metoda odpowiedzialna jest za sprawdzenie czy warunek start_rule zdefiniowany w scenariuszy dla danego węzła jest spełniony dane zawarte w *common_obj* mogą być niezbędne w tym celu
on_event(data:Dict): void	Metoda służy wywołaniu metody *run* klasy *event* dla eventu określanego przez zmienną *data*. wywoływana jest przez klasę *Analyser* po spełnieniu warunków określonych w scenariuszu dla tego Eventu

LoopNode - Jest to klasa reprezentująca jeden etap scenariusza, implementuje ona pętlę do while, stanowi implementacje klasy abstrakcyjnej *Node*. Zakłada zapętlone wykonanie wszystkich akcji zdefiniowanych w liście *actions*, aż zostanie spełniony warunek zakończenia np. określona liczbę razy. Eventy mogą być wywoływane pomiędzy akcjami lub w trakcie niektórych np *WaitAction*, zmienna *lock* jest używana w celu synchronizacji podczas wywoływania eventu. Diagram przedstawiono na rysunku 17, a szczegółowy opis w tabeli 15.

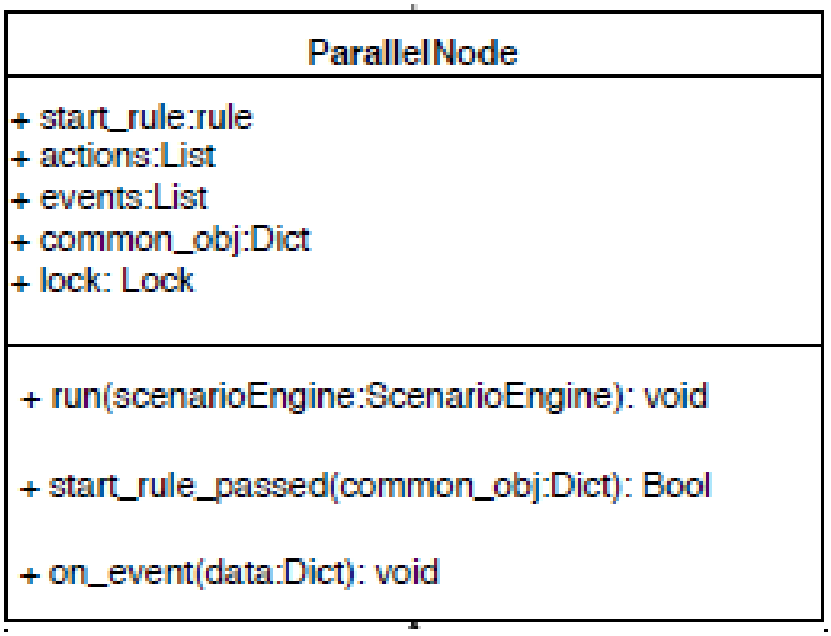


Rysunek 17: Diagram klasy *LoopNode*

Tabela 15: Przegląd metod klasy *LoopNode*

Metoda	Opis
run(scenarioEngine: ScenarioEngine): void	Metoda odpowiedzialna za uruchomienie wszystkich zaplanowanych w tym etapie scenariusza akcji. Na Początku rejestruje wszystkie eventy z listu *events* w klasie *Analuser* do sprawdzania czy nie zostały spełnione warunki ich wywołania, Po wykonaniu wszystkich akcji eventy są wyrejestrowywane. Akcję są powtarzane aż do spełnienia warunku zakończenia. Zmienna *scenarioEngine* jest przekazywana akcją w razie potrzeby
start_rule_passed(common_obj:Dict): Bool	Metoda odpowiedzialna jest za sprawdzenie czy warunek start_rule zdefiniowany w scenariuszy dla danego węzła jest spełniony dane zawarte w *common_obj* mogą być niezbędne w tym celu
end_rule_passed(common_obj:Dict): Bool	Metoda odpowiedzialna jest za sprawdzenie czy warunek end_rule zdefiniowany w scenariuszy dla danego węzła jest spełniony dane zawarte w *common_obj* mogą być niezbędne w tym celu
on_event(data:Dict): void	Metoda służy wywołaniu metody *run* klasy *event* dla eventu określanego przez zmienną *data*. wywoływana jest przez klasę *Analyser* po spełnieniu warunków określonych w scenariuszu dla tego Eventu

ParallelNode - Jest to klasa reprezentująca jeden etap scenariusza, jest ona przeznaczaną do równoległego wykonywania implementacją klasy abstrakcyjnej *Node*. Zakłada zapętlone wykonanie wszystkich akcji zdefiniowanych w liście *actions*, aż nie zostanie zatrzymana. Eventy mogą być wywoływane pomiędzy akcjami lub w trakcie niektórych np *WaitAction*, zmienna *lock* jest używana w celu synchronizacji podczas wywoływania eventu. Diagram przedstawiono na rysunku 18, a szczegółowy opis w tabeli 16.

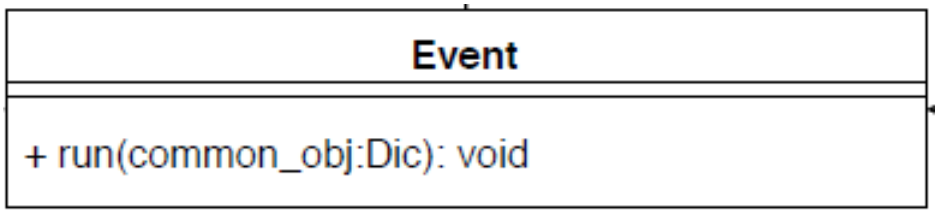


Rysunek 18: Diagram klasy *ParallelNode*

Tabela 16: Przegląd metod klasy *ParallelNode*

Metoda	Opis
run(scenarioEngine: ScenarioEngine): void	Metoda odpowiedzialna za uruchomienie wszystkich zaplanowanych w tym etapie scenariusza akcji. Na początku rejestruje wszystkie eventy z listu *events* w klasie *Analuser* do sprawdzania czy nie zostały spełnione warunki ich wywołania, Po wykonaniu wszystkich akcji eventy są wyrejestrowywane. Akcję są powtarzane aż do zakończenia wątku. Zmienna *scenarioEngine* jest przekazywana akcją w razie potrzeby
start_rule_passed(common_obj:Dict): Bool	Metoda odpowiedzialna jest za sprawdzenie czy warunek start_rule zdefiniowany w scenariuszy dla danego węzła jest spełniony dane zawarte w *common_obj* mogą być niezbędne w tym celu
on_event(data:Dict): void	Metoda służy wywołaniu metody *run* klasy *event* dla eventu określanego przez zmienną *data*. wywoływana jest przez klasę *Analyser* po spełnieniu warunków określonych w scenariuszu dla tego Eventu

Event - Jest to klasa abstrakcyjna reprezentująca event zdefiniowany w scenariuszu. Diagram przedstawiono na rysunku 19, a szczegółowy opis w tabeli 17.

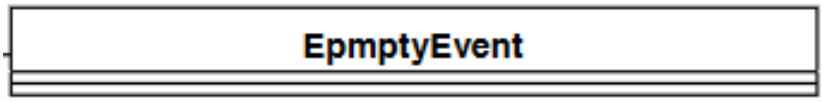


Rysunek 19: Diagram klasa abstrakcyjna *Event*

Tabela 17: Przegląd metod klasa abstrakcyjna *Event*

Metoda	Opis
run(common_obj: Dict): void	Metoda odpowiedzialna za uruchomienie wszystkich zaplanowanych dla tego eventu akcji. Zmienna *common_obj* jest przekazywana akcją w razie potrzeby

EmptyEvent - Jest to klasa impemętująca klasę *Event*. Jest to pusta klasa wykorzystywana w trakcie testów. Diagram przedstawiono na rysunku 20, a szczegółowy opis w tabeli 18.

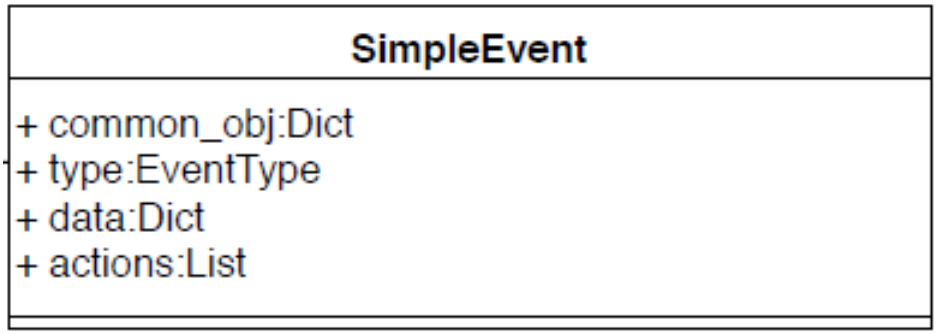


Rysunek 20: Diagram klasa *EmptyEvent*

Tabela 18: Przegląd metod klasa *EmptyEvent*

Metoda	Opis
run(common_obj: Dict): void	Metoda odpowiedzialna za uruchomienie wszystkich zaplanowanych dla tego eventu akcji. Zmienna *common_obj* jest przekazywana akcją w razie potrzeby

SimpleEvent - Jest to klasa impemętująca klasę *Event*. Jest to prosta implementacja zakłada wykonanie wszystkich akcji z listy *actions* jeżeli zostanie event zostanie wywołany przez klasę *Analyser* po spełnieniu warunków określonych przez *EventType* i zmienną *data*. Diagram przedstawiono na rysunku 21, a szczegółowy opis w tabeli 19.

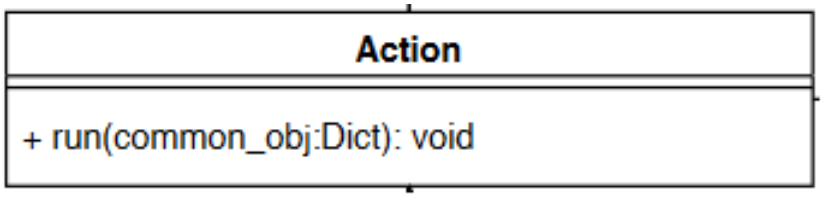


Rysunek 21: Diagram klasa *SimpleEvent*

Tabela 19: Przegląd metod klasa *SimpleEvent*

Metoda	Opis
run(common_obj: Dict): void	Metoda odpowiedzialna za uruchomienie wszystkich zaplanowanych dla tego eventu akcji. Zmienna *common_obj* jest przekazywana akcją w razie potrzeby

Action - Jest to klasa abstrakcyjna reprezentująca pojedynczą akcję zdefiniowaną w scenariuszu. Diagram przedstawiono na rysunku 22, a szczegółowy opis w tabeli 20.

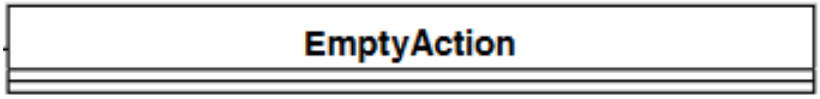


Rysunek 22: Diagram klasa abstrakcyjna *Action*

Tabela 20: Przegląd metod klasa abstrakcyjna *Action*

Metoda	Opis
run(common_obj: Dict): void	Metoda odpowiedzialna za wykonanie akcji. Zmienna *common_obj* jest przekazywana w razie potrzeby

EmptyAction - Jest to klasa impemętująca klasę *Action*. Jest to pusta klasa wykorzystywana w trakcie testów. Diagram przedstawiono na rysunku 23, a szczegółowy opis w tabeli 21.

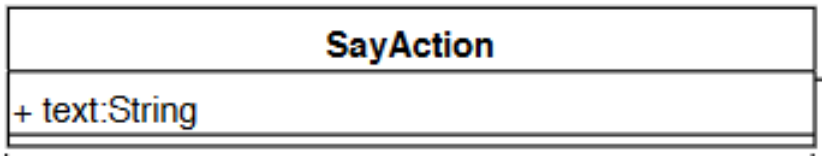


Rysunek 23: Diagram klasa *EmptyAction*

Tabela 21: Przegląd metod klasa *EmptyAction*

Metoda	Opis
run(common_obj: Dict): void	Metoda odpowiedzialna za wykonanie akcji. Zmienna *common_obj* jest przekazywana w razie potrzeby

SayAction - Jest to klasa impemętująca klasę *Action*, klasa implementuje funkcje mówienia Text to speech. Diagram przedstawiono na rysunku 24, a szczegółowy opis w tabeli 22.

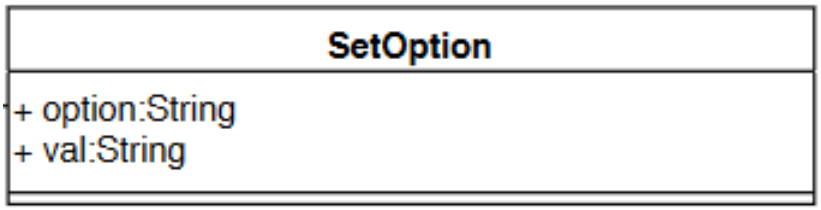


Rysunek 24: Diagram klasa *SayAction*

Tabela 22: Przegląd metod klasa *SayAction*

Metoda	Opis
run(common_obj: Dict): void	Metoda odpowiedzialna za wykonanie akcji mówienia używa do tego celu API robota. Zmienna *common_obj* jest przekazywana w w celu pozyskania sesji z robotem

SetOption - Jest to klasa impemętująca klasę *Action*, odpowiada ona za ustawianie zmiennych w "options", wykorzystywana do ustawiania zmiennych z poziomu scenariusza. Diagram przedstawiono na rysunku 25, a szczegółowy opis w tabeli 23.

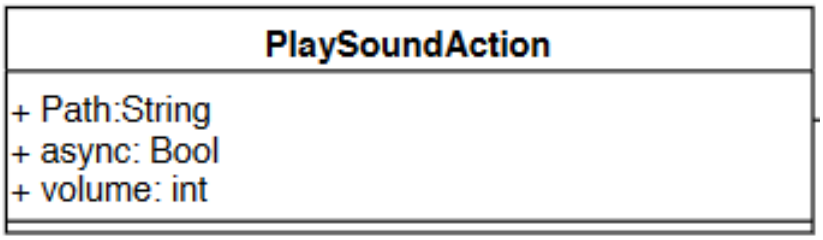


Rysunek 25: Diagram klasa *SetOption*

Tabela 23: Przegląd metod klasa *SetOption*

Metoda	Opis
run(common_obj: Dict): void	Metoda odpowiedzialna za ustawianie zmiennych w "options" przechowywane są w *common_obj*, jest to wykorzystywane do ustawiania zmiennych z poziomu scenariusza. *option* zawiera nazwę zmiennej a *val* jej wartość

PlaySoundAction - Jest to klasa impemętująca klasę *Action*, odpowiada ona za odtworzenie pliku dźwiękowego. Diagram przedstawiono na rysunku 26, a szczegółowy opis w tabeli 24.

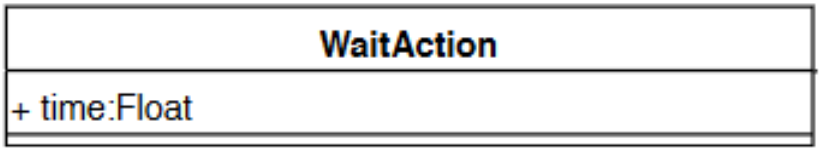


Rysunek 26: Diagram klasa *PlaySoundAction*

Tabela 24: Przegląd metod klasa *PlaySoundAction*

Metoda	Opis
run(common_obj: Dict): void	Metoda odpowiedzialna, odpowiada ona za odtworzenie pliku dźwiękowego, znajdującego się pod adresem *path*, zmienna *volume* określa poziom głośności a *async* czy funkcja ma oczekiwać na zakończenie odtwarzania czy wykonać się asynchronicznie. Zmienna *common_obj* jest przekazywana w w celu pozyskania sesji z robotem

WaitAction - Jest to klasa impemętująca klasę *Action*, odpowiada ona za oczekiwanie. Diagram przedstawiono na rysunku 27, a szczegółowy opis w tabeli 25.



Rysunek 27: Diagram klasa *WaitAction*

Tabela 25: Przegląd metod klasa *WaitAction*

Metoda	Opis
run(common_obj: Dict): void	Metoda odpowiedzialna za oczekiwanie przez czas *time*.

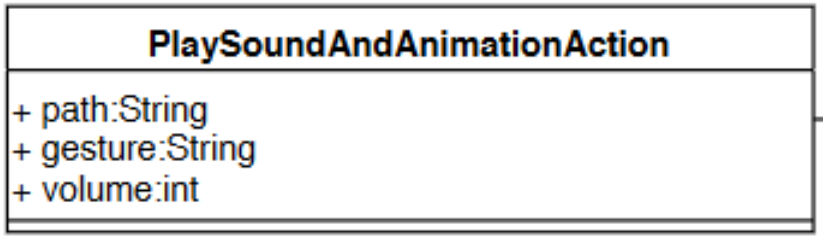
SayOnAndroidAndPlayAnimationAction - Jest to klasa impemętująca klasę *Action*, odpowiada ona za Text to speech z wykorzystaniem API Google równocześnie robot wykonuje animacje. Diagram przedstawiono na rysunku 28, a szczegółowy opis w tabeli 26.

Rysunek 28: Diagram klasa *SayOnAndroidAndPlayAnimationAction*

Tabela 26: Przegląd metod klasa *SayOnAndroidAndPlayAnimationAction*

Metoda	Opis
run(common_obj: Dict): void	Metoda odpowiada za Text to speech z wykorzystaniem API Google, zmienna *text* jest przesyłana do robota z wykorzystaniem kolejki mqtt i wypowiadana. Równocześnie robot wykonuje animacje zaplanowane w zmiennej *gesture*

PlaySoundAndAnimationAction - Jest to klasa impemętująca klasę *Action*, odpowiada ona za odtworzenie pliku dźwiękowego i równocześnie robot wykonuje animacje. Diagram przedstawiono na rysunku 29, a szczegółowy opis w tabeli 27.

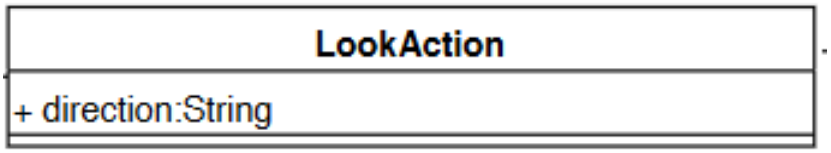


Rysunek 29: Diagram klasa *PlaySoundAndAnimationAction*

Tabela 27: Przegląd metod klasa *PlaySoundAndAnimationAction*

Metoda	Opis
run(common_obj: Dict): void	Metoda odpowiada odpowiada ona za odtworzenie pliku dźwiękowego, znajdującego się pod adresem *path*, zmienna *volume* określa poziom głośności. Równocześnie robot wykonuje animacje zaplanowane w zmiennej *gesture*

LookAction - Jest to klasa impemętująca klasę *Action*, odpowiada za skierowanie robota w danym kierunku *direction*. Diagram przedstawiono na rysunku 30, a szczegółowy opis w tabeli 28.

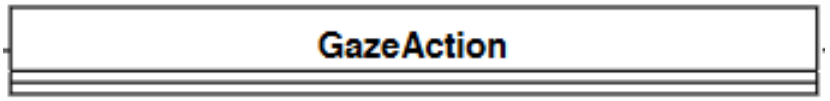


Rysunek 30: Diagram klasa *LookAction*

Tabela 28: Przegląd metod klasa *LookAction*

Metoda	Opis
run(common_obj: Dict): void	Metoda odpowiada za skierowanie robota w danym kierunku *direction*

GazeAction - Jest to klasa impemętująca klasę *Action*, odpowiada za wykonanie przez robota awersji spojrzenia. Diagram przedstawiono na rysunku 31, a szczegółowy opis w tabeli 29.

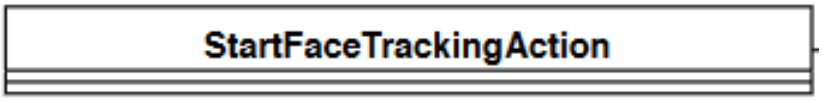


Rysunek 31: Diagram klasa *GazeAction*

Tabela 29: Przegląd metod klasa *GazeAction*

Metoda	Opis
run(common_obj: Dict): void	Metoda odpowiada za wykonanie przez robota awersji spojrzenia zgodnie z zdefiniowanym w *common_obj* modelem awersji

StartFaceTrackingAction - Jest to klasa impemętująca klasę *Action*, odpowiada za uruchomienie śledzenia twarzy użytkownika. Diagram przedstawiono na rysunku 32, a szczegółowy opis w tabeli 30.

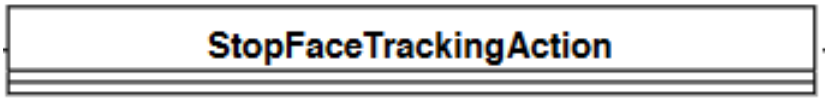


Rysunek 32: Diagram klasa *StartFaceTrackingAction*

Tabela 30: Przegląd metod klasa *StartFaceTrackingAction*

Metoda	Opis
run(common_obj: Dict): void	Metoda odpowiada za uruchomienie śledzenia twarzy użytkownika, robot będzie podążał za twarzą użytkownika.

StopFaceTrackingAction - Jest to klasa impemętująca klasę *Action*, odpowiada za zatrzymanie śledzenia twarzy użytkownika. Diagram przedstawiono na rysunku 33, a szczegółowy opis w tabeli 31.

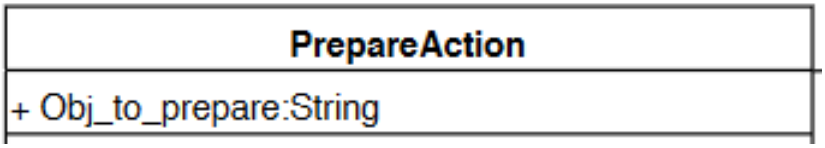


Rysunek 33: Diagram klasa *StopFaceTrackingAction*

Tabela 31: Przegląd metod klasa *StopFaceTrackingAction*

Metoda	Opis
run(common_obj: Dict): void	Metoda odpowiada za zaprzestanie śledzenia twarzy użytkownika, robot będzie przestanie podążać za twarzą użytkownika.

PrepareAction - Jest to klasa impemętująca klasę *Action*, odpowiada za przygotowanie i uruchomienie różnych modułów systemu takich jak: analiza video, rozpoznawanie pytań, rozpoznawanie mowy,... Diagram przedstawiono na rysunku 34, a szczegółowy opis w tabeli 32.

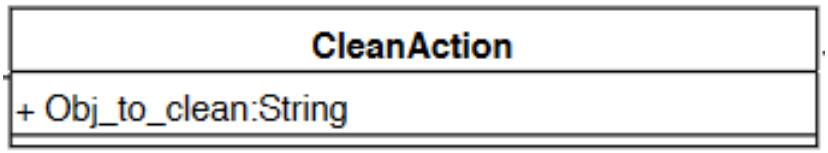


Rysunek 34: Diagram klasa *PrepareAction*

Tabela 32: Przegląd metod klasa *PrepareAction*

Metoda	Opis
run(common_obj: Dict): void	Metoda odpowiada za przygotowanie i uruchomienie modułu systemu o nazwie $Obj_to_prepare$. Referencja do modułu jest umieszczana w $common\_obj$.

CleanAction - Jest to klasa impemętująca klasę *Action*, odpowiada za bezpieczne zakończenie pracy różnych modułów systemu takich jak: analiza video, rozpoznawanie pytań, rozpoznawanie mowy,... Diagram przedstawiono na rysunku 35, a szczegółowy opis w tabeli 33.



Rysunek 35: Diagram klasa *CleanAction*

Tabela 33: Przegląd metod klasa *CleanAction*

Metoda	Opis
run(common_obj: Dict): void	Metoda odpowiada za bezpieczne zakończenie pracy modułu systemu o nazwie Obj_to_clean. Referencja do modułu jest usuwana z $common\_obj$.

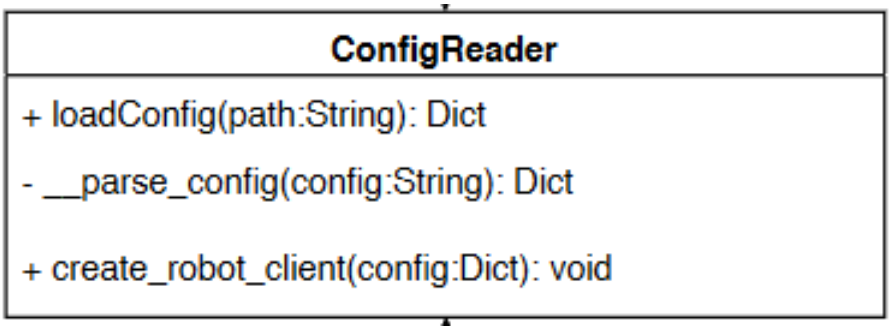
LookInDirAction - Jest to klasa impemętująca klasę *Action*, odpowiada za skierowanie robota w danym kierunku *direction*, różnica między tą klasą a klasą *LookAction* jest niewielka, ta klasa zakłada natychmiastowe zwolnienie blokady wykonania eventu podczas gdy *LookAction* robi to z 2s opóźnieniem. Diagram przedstawiono na rysunku 36, a szczegółowy opis w tabeli 34.

Rysunek 36: Diagram klasa *LookInDirAction*

Tabela 34: Przegląd metod klasa *LookInDirAction*

Metoda	Opis
run(common_obj: Dict): void	Metoda odpowiada za skierowanie robota w danym kierunku *direction*.

ConfigReader - Jest to klasa odpowiadająca za wczytanie konfiguracji i uruchomienie podstawowych modułów zgodnie z konfiguracją. Diagram przedstawiono na rysunku 37, a szczegółowy opis w tabeli 33.

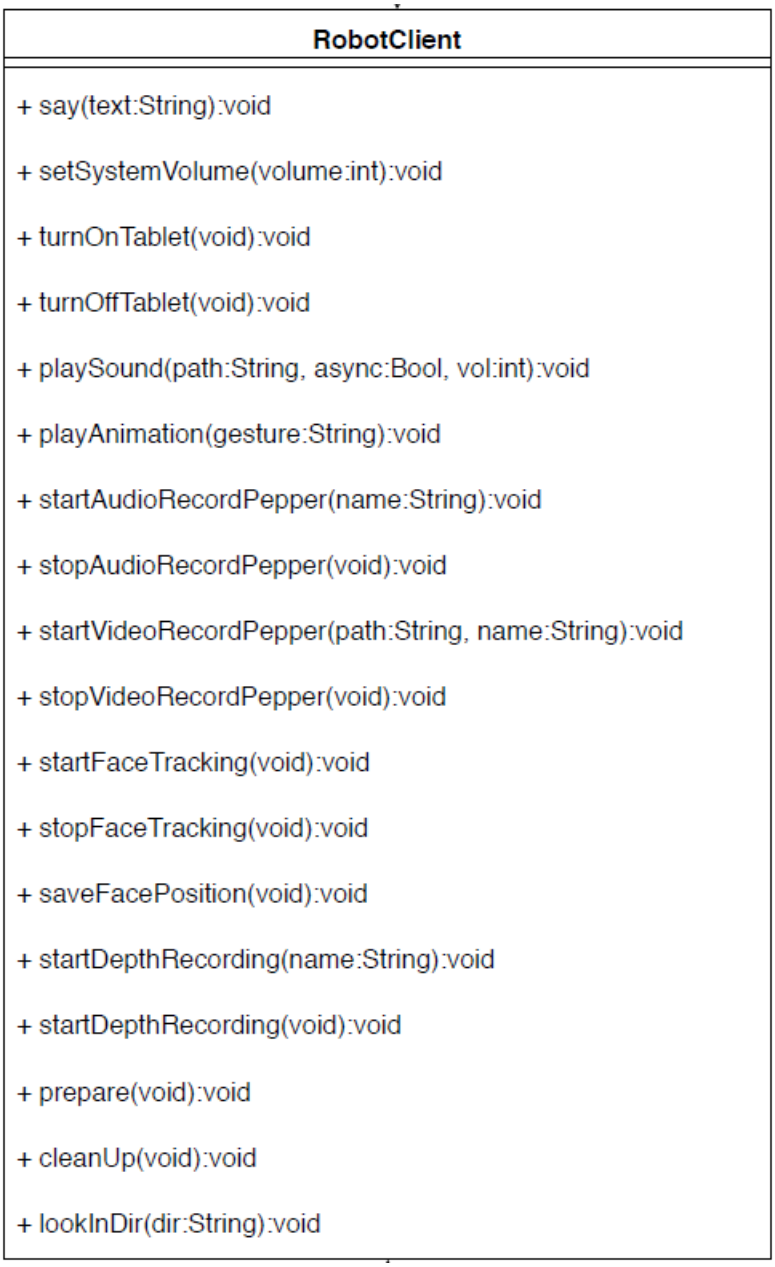


Rysunek 37: Diagram klasa *ConfigReader*

Tabela 35: Przegląd metod klasa *ConfigReader*

Metoda	Opis
loadConfig(path:String): Dict	Metoda odpowiada za wczytanie konfiguracji z pliku konfiguracyjnego w formacie json, zmienne *path* określa lokalizację pliku. Zwracany słownik jest nazywany w innych częściach systemu *common_obj* i zawiera referencje do głównych modułów systemu oraz opcji konfiguracyjnych
__parse_config(config:String): Dict	Metoda odpowiada za przeparsowanie konfiguracji wczytanej z pliku konfiguracyjnego w formacie json. Jest używana przez metodę *loadConfig*
create_robot_client(config:Dict): void	Metoda odpowiada za przygotowanie i utworzenie modułów zdefiniowanych w *config*. Jest używana przez metodę *loadConfig*

Klasa abstrakcyjna *RobotClient* - Jest to klasa abstrakcyjna odpowiedzialna za wywoływanie odpowiednich zachowań robota Pepper lub avatara. Diagram przedstawiono na rysunku 38, a szczegółowy opis w tabeli 36.



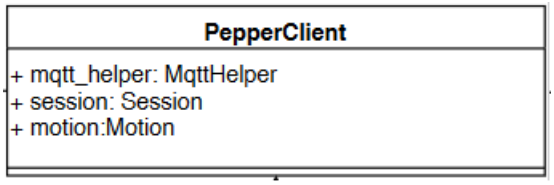
Rysunek 38: Diagram klasy abstrakcyjnej *RobotClient*

Tabela 36: Przegląd metod klasy abstrakcyjnej *RobotClient*

Metoda	Opis

say(text:String):void	metoda odpowiada za wypowiedzenie wprowadzonego tekstu *text* przez robota lub avatar.
setSystemVolume(volume:int): void	metoda zmienia poziom głośności robota lub urządzenia mobilnego.
turnOnTablet(void):void	Metoda wyłącza tablet robota
turnOffTablet(void):void	Metoda włącza tablet robota
playSound(path:String, async:Bool, vol:int): void	Metoda odpowiedzialna za odtworzenie pliku dźwiękowego, znajdującego się pod adresem *path*, zmienna *volume* określa poziom głośności a *async* czy funkcja ma oczekiwać na zakończenie odtwarzania czy wykonać się asynchronicznie.
playAnimation(gesture:String): void	Jest to metoda odpowiedzialna za wykonanie przez robota animacji zaplanowanej w zmiennej *gesture*
startAudioRecordPepper(name:String): void	Metoda odpowiedzialna jest za uruchomienie nagrywania audio i zapisanie go w miejscu *name*.
stopAudioRecordPepper(void): void	Metoda odpowiedzialna jest za zatrzymanie nagrywania audio.
startVideoRecordPepper(path:String, name:String): void	Metoda odpowiedzialna jest za uruchomienie nagrywania video i zapisanie go w miejscu *path*, pod nazwą *name*.
stopVideoRecordPepper(void):void	Metoda odpowiedzialna jest za zatrzymanie nagrywania video.
startFaceTracking(void):void	Metoda odpowiedzialna jest za uruchomienie śledzenia twarzy użytkownika, robot będzie podążał za twarzą użytkownika.
stopFaceTracking(void):void	Metoda odpowiada za zaprzestanie śledzenia twarzy użytkownika, robot będzie przestanie podążać za twarzą użytkownika.
saveFacePosition(void): void	Metoda zapisuje aktualną pozycję twarzy.
startDepthRecording(name:String):void	Metoda odpowiedzialna jest za uruchomienie nagrywania video kamerą głębi i zapisanie go w miejscu *name*.
stopDepthRecording(void):void	Metoda odpowiedzialna jest za zatrzymanie nagrywania video kamerą głębi.
prepare(void): void	Metoda przygotowuje moduł do pracy.
cleanUp(void): void	Metoda bezpiecznie kończy pracę modułu.
lookInDir(dir:String):void	Metoda odpowiada za skierowanie robota w danym kierunku *dir*

Klasa *PepperClient* - Jest to klasa odpowiedzialna za wywoływanie odpowiednich zachowań robota Pepper. Dziedziczy ona po klasie abstrakcyjnej *RobotClient*. Diagram przedstawiono na rysunku 39, a szczegółowy opis w tabeli 37.



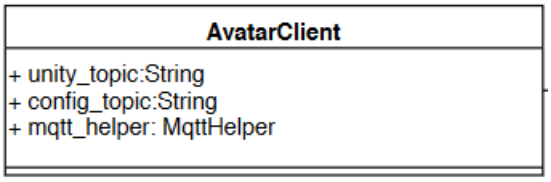
Rysunek 39: Diagram klasy *PepperClient*

Tabela 37: Przegląd metod klasy *PepperClient*

Metoda	Opis
say(text:String):void	metoda odpowiada za wypowiedzenie wprowadzonego textu *text* przez robota.
setSystemVolume(volume:int): void	metoda zmienia poziom głośności robota.
turnOnTablet(void):void	Metoda wyłącza tablet robota
turnOffTablet(void):void	Metoda włącza tablet robota
playSound(path:String, async:Bool, vol:int): void	Metoda odpowiedzialna za odtworzenie pliku dźwiękowego, znajdującego się pod adresem *path*, zmienna *volume* określa poziom głośności a *async* czy funkcja ma oczekiwać na zakończenie odtwarzania czy wykonać się asynchronicznie.
playAnimation(gesture:String): void	Jest to metoda odpowiedzialna za wykonanie przez robota animacji zaplanowanej w zmiennej *gesture*
startAudioRecordPepper(name:String): void	Metoda odpowiedzialna jest za uruchomienie nagrywania audio i zapisanie go w miejscu *name*.
stopAudioRecordPepper(void): void	Metoda odpowiedzialna jest za zatrzymanie nagrywania audio.
startVideoRecordPepper(path:String, name:String): void	Metoda odpowiedzialna jest za uruchomienie nagrywania video i zapisanie go w miejscu *path*, pod nazwą *name*.
stopVideoRecordPepper(void):void	Metoda odpowiedzialna jest za zatrzymanie nagrywania video.
startFaceTracking(void):void	Metoda odpowiedzialna jest za uruchomienie śledzenia twarzy użytkownika, robot będzie podążał za twarzą użytkownika.
stopFaceTracking(void):void	Metoda odpowiada za zaprzestanie śledzenia twarzy użytkownika, robot będzie przestanie podążać za twarzą użytkownika.
saveFacePosition(void): void	Metoda zapisuje aktualną pozycję twarzy.
startDepthRecording(name:String):void	Metoda odpowiedzialna jest za uruchomienie nagrywania video kamerą głębi i zapisanie go w miejscu *name*.
stopDepthRecording(void):void	Metoda odpowiedzialna jest za zatrzymanie nagrywania video kamerą głębi.
prepare(void): void	Metoda przygotowuje moduł do pracy.
cleanUp(void): void	Metoda bezpiecznie kończy pracę modułu.

lookInDir(dir:String):void	Metoda odpowiada za skierowanie robota w danym kierunku *dir*

Klasa *AvatarClient* - Jest to klasa odpowiedzialna za wywoływanie odpowiednich zachowań avatara, cała komunikacja z aplikacją wykonawczą odbywa się za pomocą kolejki mqtt. Dziedziczy ona po klasie abstrakcyjnej *RobotClient*. Diagram przedstawiono na rysunku 40, a szczegółowy opis w tabeli 38.



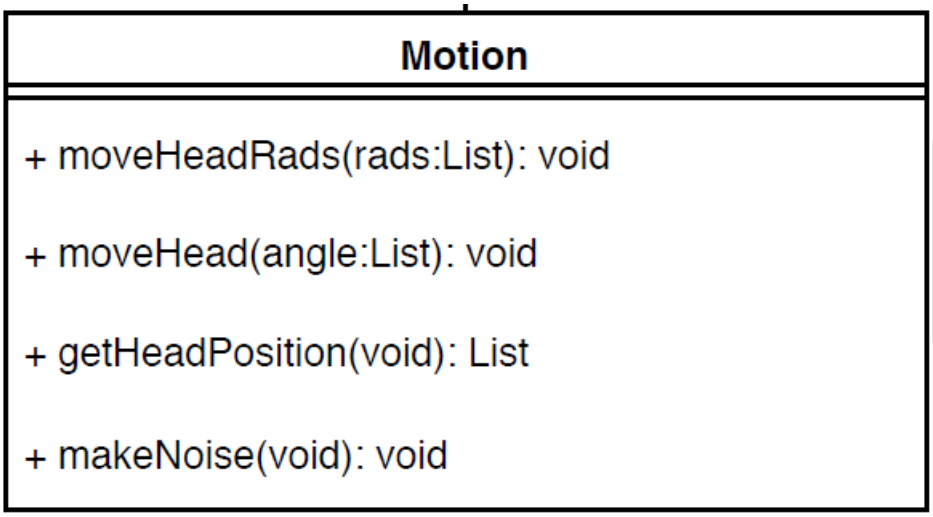
Rysunek 40: Diagram klasy *AvatarClient*

Tabela 38: Przegląd metod klasy *AvatarClient*

Metoda	Opis
say(text:String):void	metoda odpowiada za wypowiedzenie wprowadzonego tekstu *text* avatara. Text jest przesyłany przy użyciu kolejki mqtt i wypowiadany z użyciem narzędzia textToSpeech Google.
setSystemVolume(volume:int): void	metoda zmienia poziom głośności robota lub urządzenia mobilnego.
turnOnTablet(void):void	Metoda wyłącza tablet robota. Obecnie nie zaimplementowane jako bez sensowne.
turnOffTablet(void):void	Metoda włącza tablet robota. Obecnie nie zaimplementowane jako bez sensowne.
playSound(path:String, async:Bool, vol:int): void	Metoda odpowiedzialna za odtworzenie pliku dźwiękowego, znajdującego się pod adresem *path*, zmienna *volume* określa poziom głośności a *async* czy funkcja ma oczekiwać na zakończenie odtwarzania czy wykonać się asynchronicznie. Obecnie nie zaimplementowane ze względu na możliwości aplikacji avatara.
playAnimation(gesture:String): void	Jest to metoda odpowiedzialna za wykonanie przez robota animacji zaplanowanej w zmiennej *gesture*. Obecnie nie zaimplementowane ze względu na możliwości aplikacji avatara.
startAudioRecordPepper(name:String): void	Metoda odpowiedzialna jest za uruchomienie nagrywania audio i zapisanie go w miejscu *name*.
stopAudioRecordPepper(void): void	Metoda odpowiedzialna jest za zatrzymanie nagrywania audio.
startVideoRecordPepper(path:String, name:String): void	Metoda odpowiedzialna jest za uruchomienie nagrywania video i zapisanie go w miejscu *path*, pod nazwą *name*.

stopVideoRecordPepper(void):void startFaceTracking(void):void	Metoda odpowiedzialna jest za zatrzymanie nagrywania video. Metoda odpowiedzialna jest za uruchomienie śledzenia twarzy użytkownika, avatar będzie podążał za twarzą użytkownika.
stopFaceTracking(void):void	Metoda odpowiada za zaprzestanie śledzenia twarzy użytkownika, avatar będzie przestanie podążać za twarzą użytkownika.
saveFacePosition(void): void	Metoda zapisuje aktualną pozycję twarzy. Obecnie nie zaimplementowane ze względu na możliwości aplikacji avatara
startDepthRecording(name:String):void	Metoda odpowiedzialna jest za uruchomienie nagrywania video kamerą głębi i zapisanie go w miejscu *name*. Obecnie nie zaimplementowane ze względu na możliwości urządzeń mobilnych
stopDepthRecording(void):void	Metoda odpowiedzialna jest za zatrzymanie nagrywania video kamerą głębi. Obecnie nie zaimplementowane ze względu na możliwości urządzeń mobilnych
prepare(void): void	Metoda przygotowuje moduł do pracy.
cleanUp(void): void	Metoda bezpiecznie kończy pracę modułu.
lookInDir(dir:String):void	Metoda odpowiada za skierowanie avatara w danym kierunku *dir*

Motion - Jest to klasa odpowiedzialna za za wykonywanie ruchów robota wydzielona z klasy *PepperClient*. Diagram przedstawiono na rysunku 41, a szczegółowy opis w tabeli 39.

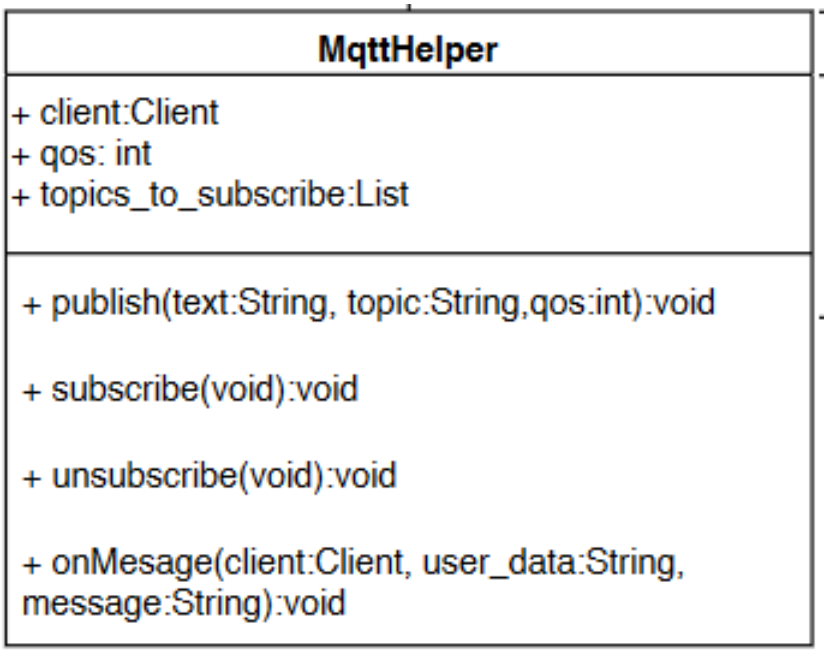


Rysunek 41: Diagram klasy *Motion*

Tabela 39: Przegląd metod klasy *Motion*

Metoda	Opis
moveHeadRads(rads:List): void	Metoda odpowiada za poruszenie głową robota o kąt podany w radianach *rads*
moveHead(angle:List): void	Metoda odpowiada za poruszenie głową robota o kąt "angle" podany w stopniach
getHeadPosition(void): List	Metoda odpowiada za pobranie informacji o pozycji głowy robota zwraca ją pod postacią listy kątów "pitch" i "yaw"
makeNoise(void): void	Metoda odpowiada za wykonywanie przez robota niewielkich ruchów głową mających symulować podobne ruchy u człowieka

MqttHelper - Jest to klasa odpowiedzialna za komunikację z wykorzystanie kolejki mqtt, Wykorzystuje bibliotekę paho Mqtt. Diagram przedstawiono na rysunku 42, a szczegółowy opis w tabeli 40.



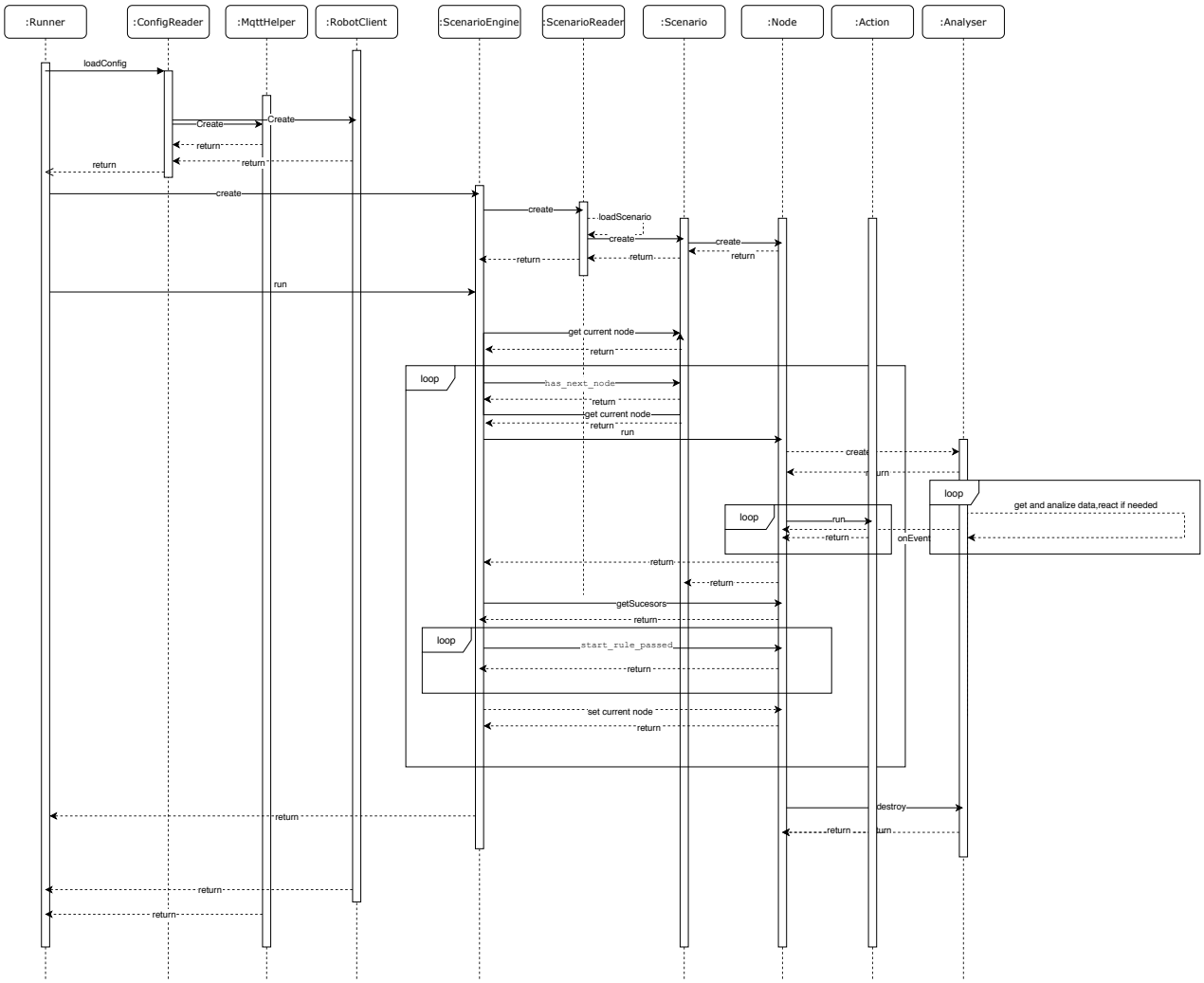
Rysunek 42: Diagram klasy *MqttHelper*

Tabela 40: Przegląd metod klasy $MqttHelper$

Metoda	Opis
publish(text:String, topic:String, qos:int): void	Metoda odpowiada za publikowanie wiadomości *text* na kolejce mqtt w kanale *topic* z quality of service równym *qos*
subscribe(void): void	Metoda odpowiada za za subskrybowanie się na kanały *topics_to_subscribe*.
unsubscribe(void):void	Metoda odpowiada za od subskrybowanie się z kanałów *topics_to_subscribe*.
onMesage(client:Client, user_data:String, message:String): void	Metoda odpowiada za odbieranie przychodzących na za subskrybowane kanały wiadomości i odpowiednie na nie zareagowanie.

2.3 Diagram sekwencji

Diagram 43 przedstawia sekwencje wywołań funkcji podczas przebiegu scenariusza. Przedstawiony diagram został uproszczony w miejscu modułu analizującego dane, w celu poprawienia czytelności, w szczególności moduł ten jest tworzony jedynie raz podczas działania programu przez węzeł przygotowujący moduły, zgodnie z definicją w scenariuszu.



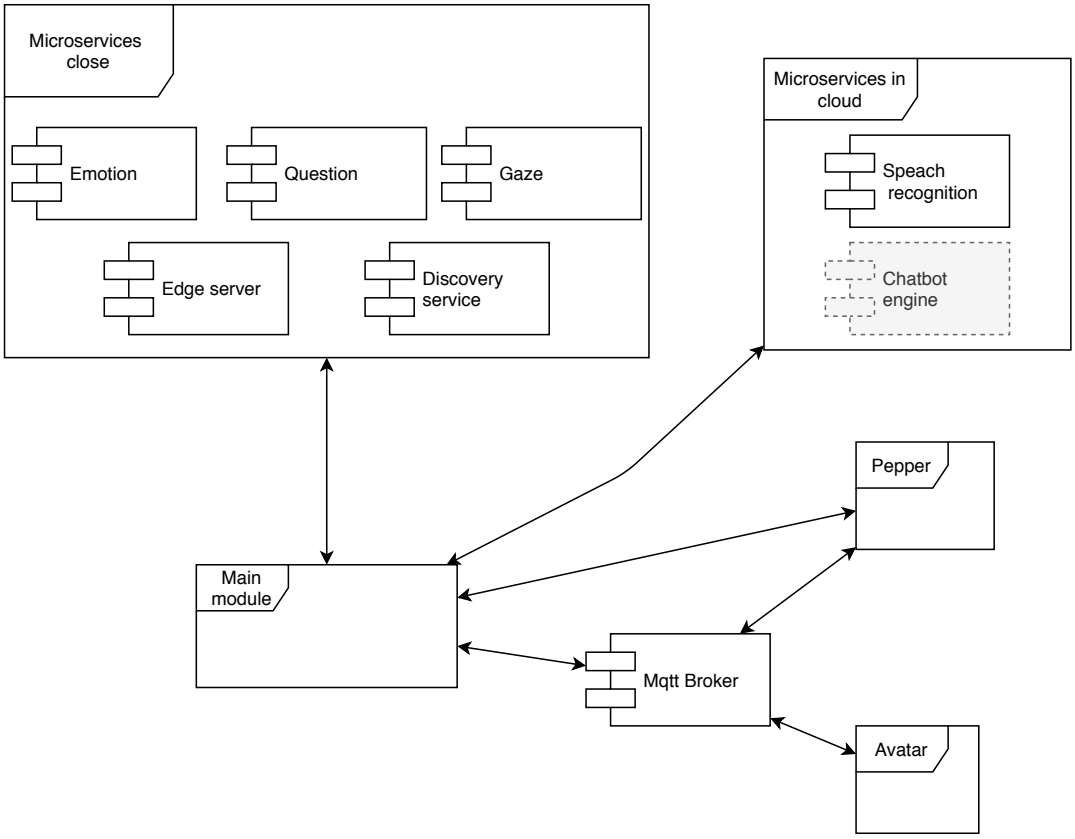
Rysunek 43: Diagram sekwencji pracy systemu

Moduł *ScenarioEngine* zarządza całym cyklem wykonania scenariusza. Przebieg scenariusza: najpierw inicjalizowany jest moduł *Runner*, który następnie powołuje do życia *ConfigReader* i wczytuje konfiguracje systemu oraz inicjuje niezbędne modułu *RobotClient* i *MqttHelper* o ile tak zostało zdefiniowane w pliku konfiguracyjnym. Następnie moduł *ScenarioEngine* jest inicjowany on wywołuje moduł *ScenarioReader* w celu wczytania z pliku scenariusza i otrzymania modułu *Scenario*. Moduł *Scenario* zawiera graf złożony z elementów *Node*. Następnie *ScenarioEngine* inicjuje wykonanie scenariusza pobierając obecny węzeł *curentNode*. Następnie sprawdzamy czy obecny węzeł ma dzieci, system zakłada istnienie co najmniej 2 węzłów startowego root i co najmniej jednego dziecka. Ponownie pobieramy obecny węzeł, wykonujemy go używając metody *run*, przeważnie pierwszy węzeł inicjalizuje inne moduły tu zostało to pokazane na przykładzie inicjalizacji modu-

łu analizatora. Moduł *Analyser* został uproszczony do jednej pętli pobierającej i analizującej dane która może wywoływać eventy na module node. Pobieranie i wyodrębnienie danych wyższego poziomu jest obsługiwane przez inne moduły (*AudioSource* i *VideoSource* do pobierania nie przewożonych danych, *EmotionDetection*,*QuestionDetection*,*GazeEstimation*,*speechRecognition* do przeważania danych. Dane są przetwarzane w sposób ciągły i spływają na kolejki danych do analizy modułu *Analyser*), jednak dodatkowe moduły spowodowały by kompletną nieczytelność schematu. Węzeł wywołuje w pętli funkcje run dla wszystkich zdefiniowanych dla niego w scenariuszy akcji *Action*, akcje mogą wykorzystywać moduły *MqttHelper* i *RobotClient* w celu wykonania. Pomiędzy akcjami jak i w czasie niektórych akcji moduł *Analyser* jest w stanie wywołać metodę *onEvent* doprowadzając do przerwy w wykonywaniu scenariusza i wykonania akcji zaplanowanych dla danego eventu. Następnie moduł *ScenarioEngine* pobiera dzieci obecnego węzła sprawdza które zwracają prawdę dla *start_rule_passing* i wybiera z pośród nich najlepszy, ustawia go jako *current_node* ta pętla powtarza się do skończenia się węzłów grafie scenariusza czyli do momentu dojścia do liścia. Finalnie wszystkie uruchomione moduły są zamykane i system kończy pracę.

3 Architektura systemu - implementacja

Schemat systemu (rys. 44) przestawia szczegóły implementacji systemu w szczególności wszystkie niezbędne pod systemy współpracujące z głównym systemem. Wspomniane do tej pory jako zewnętrzne serwisy realizujące zadania związane z analizą nie przetworzonych danych w celu pozyskania danych wysokiego poziomu(np. emocji użytkownika na podstawie wideo).



Rysunek 44: Diagram Implementacji systemu

- Main module - opisany jest szeroko w poprzednich rozdziałach, dlatego na diagramie 44 został jedynie zaznaczony bez wyszczególniania modułów. Komunikuje się z innymi modułami przy użyciu Mqtt brokera (moduły robota/ avatara) z robotem nawiązuje bezpośrednie połączenie z wykorzystaniem natywnego API. Łączy się z modułami oznaczonymi jako Microservices close przy użyciu Http, moduł speech recognition wykorzystuje API Google Cloud.
- Mqtt Broker - jest to broker mqtt odpowiada on za przekazywanie większości wiadomości między głównym modułem a robotem lub aplikacją avatara. Broker może być uruchomiony w dowolnym miejscu na świecie o ile można na wiązać z nim połączenie sieciowe jednak ze względu na czas przesyłu danych jest on uruchamiany w tej samej sieci lokalnej co moduł główny i robot.
- Microservices close - są to aplikacje stworzone w architekturze mikro serwisowej w oparciu o framework Netflix OSS. Mikro serwisy odpowiedzialne są za analizą nie przetworzonych danych w celu pozyskania danych wysokiego poziomu. Dodatkowo zaznaczona serwisy realizujące funkcje API Gateway (edge server - zuul) i Discowery(eureka).
- Microservices in cloud - oznaczają zewnętrzne serwisy dostarczane jako usługę Speech re-

cognition dostarcz Google, moduł chatbot jest planowany na przyszłość przy użyciu Google Dialogflow.

- Avatar i Pepper - reprezentują moduły wykonawcze czyli robota Pepper i aplikację avatara na urządzenie mobilne.

3.1 Szczegóły implementacji - Microservices close

Serwisy zostały strwożone w oparciu o framework Netflix OSS, w sposób bez stanowy tak by można je łatwo skalować horyzontalnie z użyciem takich narzędzi jak docker swarm. Edge server - jest to serwer zuul z pakietu Netflix OSS zaimplementowany w języku Java dostosowany i skonfigurowany do potrzeb całej aplikacji pełni on role API Gateway, podobnie sytuacja wygląda w przypadku Discovery service jest to serwer Eureka z pakietu Netflix OSS.

Moduł Emotion - jest zaimplementowany w języku Python przy użyciu biblioteki Flask. Aplikacja jest hostowana przez serwer Gunicorn z 4 workerami. Moduł rejestruje się w Discovery service by być dostępny dla klientów z użyciem biblioteki pyeureka. Aplikacja odpowiada na 3 endpointach "/info" z informacjami o serwerze, "/emotions" metoda GET jest wykorzystywana jedynie do testów, metoda POST jest właściwą metodą odpowiedzialną za wykrywanie emocji. Jako wejście przyjmuje obraz w skali szarości zwraca informacje o wykrytej emocji wraz z jej prawdopodobieństwem. Obraz powinien zawierać jedynie twarz wykrytą wcześniej przy użyciu odpowiedniego detektora. Za sam proces wykrywania emocji odpowiedzialna jest siec neuronowa napisana w frameworku tensorflow, rozpoznaje ona emocje średnio a 86% skutecznością.

Moduł Question - jest zaimplementowany w języku Python przy użyciu biblioteki Flask. Aplikacja jest hostowana przez serwer Gunicorn z 4 workerami. Moduł rejestruje się w Discovery service by być dostępny dla klientów z użyciem biblioteki pyeureka. Aplikacja odpowiada na 3 endpointach "/info" z informacjami o serwerze, "/questions" metoda GET jest wykorzystywana jedynie do testów, metoda POST jest właściwą metodą odpowiedzialną za wykrywanie pytań. Jako Wejście przyjmuje 3 sekundowy fragment nagrania audio w formacie mono z próbkowaniem 16KHz, nagranie może być spakowane z użyciem gzip w celu oszczędzenia przepustowości łącza. Samo rozpoznawanie odbywa się z użyciem konwokacyjnej sieci neuronowej zaimplementowanej w frameworku tensorflow w oparciu o architekturę Xception. Dźwięk jest przekształcany do postaci mel spektrogramu i w tej postaci przetwarzany przez sieć. Sieć określa czy zadano pytanie w danym fragmencie czy nie.

Moduł Gaze - jest zaimplementowany w języku Python przy użyciu biblioteki Flask. Aplikacja jest hostowana przez serwer Gunicorn z 4 workerami. Moduł rejestruje się w Discovery service by być dostępny dla klientów z użyciem biblioteki pyeureka. Aplikacja odpowiada na 3 endpointach "/info" z informacjami o serwerze, "/gaze" metoda GET jest wykorzystywana jedynie do testów, metoda POST jest właściwą metodą odpowiedzialną za wykrywanie kierunku spojrzenia. Jako wejście przyjmuje obraz w skali szarości. Obraz powinien zawierać jedynie twarz wykrytą wcześniej przy użyciu odpowiedniego detektora. Kierunek spojrzenia jest rozpoznawany przy użyciu biblioteki OpenFace.

Wszystkie moduły są przystosowane do współpracy z ELK stack w celu zbierania logów i monitorowania działania systemu. Sam moduł ELK nie został wyszczególniony na diagramie 44 ze względu na to że nie jest konieczny do uruchomienia systemu i pełni jedynie role pomocniczą.

3.2 Szczegóły implementacji - Avatar

Jest to aplikacja napisana na urządzenie z systemem android, za wyświetlanie i ruchy avatara odpowiada moduł napisany w języku unity zintegrowany z aplikacją android jako biblioteka. Aplikacja android odpowiedzialna jest za połączenie z Brokerm Mqtt i przetwarzanie poleceń napływających na odpowiednie tematy, odpowiada także za obsługę strumieni audio i video zapis ich na urządzeniu mobilnym i przesyłanie ich do modułu głównego do analizy, wypowiadanie przez avatara przesłanych z modułu głównego treści przy użyciu TextToSpeech google/android. Tematy wykorzystywane w aplikacji "config/" do obsługi konfiguracji i włączania wyłączania śledzenia twarzy użytkownika przez avatara, "update/" do przesyłania danych wideo/ audio do modułu głównego. "pepper/video" do włączania/wyłączania przetwarzania i zapisu strumienia wideo i audio, "unity/" do odbierania poleceń z modułu głównego co do ruchów avatara, "pepper/textToSpeech" do odbierania wiadomości wypowiadanych przez avatara.

3.3 Szczegóły implementacji - Pepper

Robot pepper wykorzystuje dwie metody komunikacji bezpośrednio z głównym modułem aplikacji z użyciem natywnego API dostarczanego przez producenta, oraz z użyciem brokera mqtt którego w tym wypadku sam hostuje. W drugim wypadku na tablecie robota działa aplikacja na urządzenie z systemem android będąca uproszczoną wersją aplikacji Avatar nie zawiera ona modułów: odpowiedzialnych za wyświetlanie avatara, przesyłanie audio i video do modułu głównego, analizę wideo(śledzenie twarzy). Jej głównym zadaniem jest wykonanie nagrań w wysokiej jakości z w miarę stabilnej perspektywy oraz wypowiadanie słów przy użyciu TextToSpeech google/android. Do wypowiadania słów nie wykorzystano bibliotek robota ze względu na większe możliwości wyboru języka jak i głosu na urządzeniu z systemem android oraz na początkowe problemy/brak języka polskiego d robocie.

4 Scenariusz - składnia

Scenariusze są zapisywane w formacie json, na listingu 1 widzimy minimalistyczny przykład zawierający pojedynczy węzeł root.

```
1 {
2   "root" : {
3     "type" : "sync",
4     "start_rule" : "always()",
5     "end_rule" : "child_ready()",
6     "events" : [],
7     "actions" : [],
8     "children" : []
```

```
  }
}
```

Listing 1: najprostszy przykład

Pierwsze pole *type* określa typ węzła Node w przykładzie jest używany *sync* odpowiadający *SyncNode*, możemy tu wpisać *loop* albo *parallel* i otrzymamy w tedy węzeł pożądanego typu. następnie jest warunek startu w tym przypadku wywołujący funkcje *always*(). Funkcja ta zgodnie z nazwą zwraca zawsze prawdę, warunek końca korzysta z funkcji $child_r eady()$ działa ona tak samo jak *always*(). W miejscu warunku początku i końca można podać wyrażenie które ewaluuje się do prawdy bądź fałszu. Następnie mamy pole *events* zawiera ono listę evetów. Kolejne jest pole *actions* zawiera ono listę akcji, ostatnie pole *children* będzie zawierało listę dzieci danego węzła, każde z dzieci też jest węzłem i może zawierać nie pustą listę swoich dzieci, a dzieci dzieci mogą mieć nie puste listy swoich dzieci itd. aż dojdziemy do liścia.

Na listingu 2 widzimy przykładową definicje pola *actions* z listingu 1. W liscie definiujemy kolejne obiekty *Action*, każdy taki obiekt zostanie przekształcony do odpowiedniego obiektu klasy dziedziczącej po *Action* na podstawie pola *type*, pozostałe pola podawane są w zależności od pola *type* nadmiarowe pole zostanie zignorowane brak któregokolwiek z wymaganych pul zostanie zasygnalizowany błędem. W pliku z klasą *ScenarioReader* znajduje się kurtki fragment kodu pozwalający sprawdzić czy przygotowany scenariusz ma prawidłową składnie. Pierwsza z akcji o typie *setOptionAction* Ustawi zmienną o nazwie "contact_mode" na wartość true, druga akcja uruchomi śledzenie twarzy przez robota lub avatara.

```

   "actions" : [
       {
               "Action": {
                 "type": "setOptionAction",
                 "var": "contact_mode",
                 "val": true
               }
         },
         {
           "Action": {
             "type": "startFaceTrackingAction"
           }
         }

   ]
```

Listing 2: Przykład definicji akcji

Na listingu 3 widzimy przykładową definicje pola *events* z listingu 1. W liście definiujemy kolejne obiekty *Event*, każdy taki obiekt zostanie przekształcony do odpowiedniego obiektu klasy

dziedziczącej po *Event*. event zawiera pole *type* definiuje ono typ eventu wykorzystywane jest w trakcie sprawdzania czy należy wywołać konkretny event. W tym przypadku event ma reagować na wykrycie odpowiedniej emocji zdefiniowanej w polu *value*, eventy mogą być wywoływane z pewną częstotliwością w tym przypadku nie częściej niż co 15 sekund. Ostatnie pole *actions* działa dokładnie tak samo jak dla węzła i jest zaprezentowane na listingu 2.

```
"events" : [
              {
                "Event": {
                  "type" : "emotion_occurred",
                  "separation": 15,
                  "value" : "sad",
                  "actions" : []
                }
              },
             ]
```

Listing 3: Przykład definicji eventu

Na listingu 4 widzimy przykładową definicje pola *children* z listingu 1. Wszystkie pola zostały opisane na listingach 1 - 3, na listingu 4 możemy zauważyć mechanizm zagnieżdżania dzieci na kolejnych poziomach węzłów.

```
"children" : [
{
   "type" : "sync",
   "start_rule" : "always()",
   "end_rule" : "child_ready()",
   "events" : [],
   "actions" : [],
   "children" : [
       {
       "type" : "sync",
       "start_rule" : "always()",
       "end_rule" : "child_ready()",
       "events" : [],
       "actions" : [],
       "children" : [
               {
           "type" : "sync",
           "start_rule" : "always()",
           "end_rule" : "child_ready()",
           "events" : [],
           "actions" : [],
           "children" : []
```

```
          }
        ]
      }
    ]
  },
  {
    "type" : "sync",
    "start_rule" : "always()",
    "end_rule" : "child_ready()",
    "events" : [],
    "actions" : [],
    "children" : []
  }
]
```

Listing 4: Przykład definicji dzieci

Rozmiar list w polach *actions*, *events*, *children* jest teoretycznie nie ograniczony, i limitowany jedynie ilością dostępnej pamięci.