
Taxi Simulator Documentation

Release 0.2.0

Javi Palanca

Apr 03, 2018

Contents

1	Taxi Simulator	3
1.1	Features	3
1.2	Credits	3
2	Installation	5
2.1	Stable release	5
2.2	From sources	5
3	Quickstart	7
3.1	Usage	7
3.2	Command-line interface	7
3.3	Graphical User Interface	11
3.4	Loading Scenarios	14
4	Developing New Strategies	17
4.1	Introduction	18
4.2	Agent Foundations	24
4.3	How to implement your own strategies	28
4.4	How to Implement New Strategies (Level 1) – Recommendations	36
5	API Documentation	39
5.1	taxi_simulator package	39
6	Contributing	57
6.1	Types of Contributions	57
6.2	Get Started!	58
6.3	Pull Request Guidelines	59
6.4	Tips	59
7	Credits	61
7.1	Development Lead	61
7.2	Contributors	61
8	History	63
8.1	0.3 ()	63
8.2	0.2 (2017-11-15)	63
8.3	0.1.3 (2017-11-15)	63

8.4	0.1.1 (2017-11-14)	63
8.5	0.1.0 (2017-11-03)	63
9	Indices and tables	65
	Bibliography	67
	Python Module Index	69

Contents:

CHAPTER 1

Taxi Simulator

Agent-based taxi simulator to test strategies

- Free software: MIT license
- Documentation: <https://taxi-simulator.readthedocs.io>.

1.1 Features

- Strategy pattern
- Continuous simulator
- Load scenarios
- Multi-agent system built with [SPADE](#)
- XMPP communications

1.2 Credits

This package was created with [Cookiecutter](#) and the [audreyr/cookiecutter-pypackage](#) project template.

2.1 Stable release

To install Taxi Simulator, run this command in your terminal:

```
$ pip install taxi_simulator
```

This is the preferred method to install Taxi Simulator, as it will always install the most recent stable release.

If you don't have [pip](#) installed, this [Python installation guide](#) can guide you through the process.

2.2 From sources

The sources for Taxi Simulator can be downloaded from the [Github repo](#).

You can either clone the public repository:

```
$ git clone git://github.com/javipalanca/taxi_simulator
```

Or download the [tarball](#):

```
$ curl -OL https://github.com/javipalanca/taxi_simulator/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```


Table of Contents

- *Quickstart*
 - *Usage*
 - *Command-line interface*
 - * *Running a simulation from the command-line*
 - * *Saving the simulation results*
 - * *Advanced options*
 - *Graphical User Interface*
 - *Loading Scenarios*

3.1 Usage

Using Taxi Simulator is as easy as running the application in a command line. There are two use modes: a command-line interface and a graphical web-based view. You can run simulations using only the command line or using the more easy and intuitive graphical user interface. Running Taxi Simulator without your own developed coordination strategies is possible since the application comes with a set of default strategies. Let's explore how to use both user interfaces.

3.2 Command-line interface

After installing Taxi Simulator open a command-line and type `taxi_simulator`. This starts a simulator without any options and runs the coordinator agent. The console will output the default logging information and you can

terminate the simulator by pressing Ctrl+C. When you terminate the simulator the results of the simulations are printed.

```
$ taxi_simulator
INFO:root:Starting Taxi Simulator
INFO:CoordinatorAgent:Coordinator agent running
INFO:CoordinatorAgent:Web interface running at http://127.0.0.1:9000/app
INFO:root:Creating 0 taxis and 0 passengers.
INFO:RouteAgent:Route agent running
WARNING:RouteAgent:Could not load cache file.

^C
INFO:root: Terminating... (0.0 seconds elapsed)
Simulation Results

| Avg Waiting Time | Avg Total Time | Simulation Time | Simulation Finished |
| 0 | 0 | 0 | True |

Passenger stats

| name | waiting_time | total_time | status |

Taxi stats

| name | assignments | distance | status |
```

However, if you don't use some options when running the simulator there will be no default taxis nor passengers. That's why stats are empty. To run a simulation with some parameters you can use the command-line interface options.

To show these options you can enter the `--help` command:

```
$ taxi_simulator --help

Usage: taxi_simulator [OPTIONS]

    Console script for taxi_simulator.

Options:
  -n, --name TEXT                Name of the simulation execution.
  -o, --output TEXT              Filename to save simulation results.
  -of, --oformat [json|excel]   Output format used to save simulation results.
  ↪ (default: json)
  -mt, --max-time INTEGER        Maximum simulation time (in seconds).
  -r, --autorun                  Run simulation as soon as the agents are ready.
  -t, --taxi TEXT                Taxi strategy class (default:
  ↪ AcceptAlwaysStrategyBehaviour).
  -p, --passenger TEXT           Passenger strategy class (default:
  ↪ AcceptFirstRequestTaxiBehaviour).
  -C, --coordinator TEXT         Coordinator strategy class (default:
  ↪ DelegateRequestTaxiBehaviour).
  --port INTEGER                 Web interface port (default: 9000).
  -nt, --num-taxis INTEGER       Number of initial taxis to create (default: 0).
  -np, --num-passengers INTEGER  Number of initial passengers to create (default: 0).
  --scenario TEXT                Filename of JSON file with initial scenario.
  ↪ description.
```

```

-cn, --coordinator-name TEXT    Coordinator agent name (default: coordinator).
--passwd TEXT                    Coordinator agent password (default: coordinator_
↪passwd).
-bp, --backend-port INTEGER      Backend port (default: 5000).
-v, --verbose                    Show verbose debug level: -v level 1, -vv level 2, -
↪vvv level 3, -vvvv level 4
--help                          Show this message and exit.

```

3.2.1 Running a simulation from the command-line

To run a quick simulation from the command-line you need to set up a few arguments: the number of taxis, the number of passengers and (optionally) the maximum time of simulation. The argument `--num-taxis` (or `-nt`` initializes the specified number of taxis in random positions of the map. The argument `--num-passengers` (or `-np`` initializes the specified number of passengers in random positions of the map and with a random destination for each one. If you want limit the simulation time you can use the `--max-time` argument (or `-mt`) to set the maximum number of seconds after which the simulation will end. Finally, the `--autorun` argument (or `-r`) automatically runs the simulation (this argument is important when you are not using the graphical interface, since it is the only way to start the simulation).

Example:

```

$ taxi_simulator --num-taxis 2 --num-passengers 2 --max-time 60 --autorun
INFO:root:Starting Taxi Simulator
INFO:CoordinatorAgent:Coordinator agent running
INFO:CoordinatorAgent:Web interface running at http://127.0.0.1:9000/app
INFO:root:Creating 2 taxis and 2 passengers.
INFO:RouteAgent:Route agent running
INFO:CoordinatorAgent:Simulation started.
INFO:PassengerAgent:Passenger michelle08 asked for a taxi to [39.469057, -0.406452].
INFO:PassengerAgent:Passenger schapman asked for a taxi to [39.465762, -0.382746].
INFO:TaxiAgent:Taxi stevencortez sent proposal to passenger michelle08
INFO:TaxiAgent:Taxi austin05 sent proposal to passenger michelle08
INFO:PassengerAgent:Passenger michelle08 accepted proposal from taxi stevencortez@127.
↪0.0.1
INFO:PassengerAgent:Passenger michelle08 refused proposal from taxi austin05@127.0.0.1
INFO:TaxiAgent:Taxi stevencortez on route to passenger michelle08
INFO:PassengerAgent:Passenger michelle08 informed of status: 11
INFO:PassengerAgent:Passenger michelle08 waiting for taxi.
INFO:TaxiAgent:Taxi stevencortez has arrived to destination.
INFO:PassengerAgent:Passenger schapman asked for a taxi to [39.465762, -0.382746].
INFO:TaxiAgent:Taxi austin05 sent proposal to passenger schapman
INFO:PassengerAgent:Passenger schapman accepted proposal from taxi austin05@127.0.0.1
INFO:TaxiAgent:Taxi austin05 on route to passenger schapman
INFO:PassengerAgent:Passenger schapman informed of status: 11
INFO:PassengerAgent:Passenger schapman waiting for taxi.
INFO:TaxiAgent:Taxi stevencortez has picked up the passenger michelle08@127.0.0.1.
INFO:PassengerAgent:Passenger michelle08 informed of status: 12
INFO:PassengerAgent:Passenger michelle08 in taxi.
INFO:TaxiAgent:Taxi stevencortez has arrived to destination.
INFO:TaxiAgent:Taxi stevencortez has dropped the passenger michelle08@127.0.0.1 in
↪destination.
INFO:PassengerAgent:Passenger michelle08 informed of status: 22
INFO:PassengerAgent:Passenger michelle08 arrived to destination after 10.8725750446
↪seconds.
INFO:TaxiAgent:Taxi austin05 has arrived to destination.
INFO:TaxiAgent:Taxi austin05 has picked up the passenger schapman@127.0.0.1.

```

```

INFO:PassengerAgent:Passenger schapman informed of status: 12
INFO:PassengerAgent:Passenger schapman in taxi.
INFO:TaxiAgent:Taxi austin05 has arrived to destination.
INFO:TaxiAgent:Taxi austin05 has dropped the passenger schapman@127.0.0.1 in_
↪destination.
INFO:PassengerAgent:Passenger schapman informed of status: 22
INFO:PassengerAgent:Passenger schapman arrived to destination after 22.221298933_
↪seconds.
INFO:root:
Terminating... (22.7 seconds elapsed)
INFO:CoordinatorAgent:Stopping taxi stevencortez
INFO:CoordinatorAgent:Stopping taxi austin05
INFO:CoordinatorAgent:Stopping passenger michelle08
INFO:CoordinatorAgent:Stopping passenger schapman
Simulation Results

```

Avg Total Time	Avg Waiting Time	Simulation Time	Max Time	Simulation_
↪Finished				
22.69	16.55	22.6766	60	True
↪				

Passenger stats

name	total_time	waiting_time	status
michelle08@127.0.0.1	22.685	10.8726	PASSENGER_IN_DEST
schapman@127.0.0.1	22.6845	22.2213	PASSENGER_IN_DEST

Taxi stats

name	assignments	distance	status
stevencortez@127.0.0.1	1	4835.1	TAXI_WAITING
austin05@127.0.0.1	1	7885.2	TAXI_WAITING

By analyzing the output of the simulation we can see what events have occurred and how the simulation has been developed. There were created two taxis (stevencortez and austin05) and two passengers (michelle08 and schapman). After the negotiation provided by the default strategies included in Taxi Simulator, taxi stevencortez was assigned to passenger michelle08 and taxi austin05 was assigned to passenger schapman. After 22 seconds of simulation both passengers were delivered in their destinations and taxis are free again to attend more passenger requests.

The output of the simulation also shows some statistics of the simulation, with the *Average Total Time*, which represents the average time of passengers from the moment they request a taxi until they are delivered to their destination, and the *Average Waiting Time*, which is the average time of passengers from requesting a taxi to being picked up. This information is also shown for each passenger along with their status at the end of the simulation.

In the case of taxis, the shown information includes the number of assignments of each taxi (how many passengers it has delivered), the total distance it has traveled and its final status.

This information is going to be useful for the development of new strategies that improve the system balancing or for debugging errors if a taxi or a passenger gets stuck or any other unexpected situation occurs.

3.2.2 Saving the simulation results

If you want to store the results of simulation in a file you may use the `--output` option (or `-o`) to specify the name of the file where the simulation results will be saved. The `--oformat` (`-of`) allows you to choose the output format between json (default) or excel. It is also useful to use the `--name` (or `-n`) to name the simulation.

Example:

```
$ taxi_simulator --name "My Simulation" --output results.xls --oformat excel
```

3.2.3 Advanced options

There are other options that are less common and that you probably don't need to use very often. These are options that allow you to change connection ports or default name and password of the coordinator agent. Use them only if there is a port or name conflict.

The last but no less important option is the verbosity option. It allows you to specify how verbose you want the simulator to be. The number of `v` letters you pass to the option indicates the level of verbosity (e.g. `-v` is **DEBUG** verbosity and `-vvvv` is the highest level of verbosity where the internal messages of the platform are shown).

Note: You may have noticed that we haven't discussed three very important options that are: `--taxi`, `--passenger`, and `--coordinator`. These options are used to inject new strategies to the simulator and we'll be discussed in a later chapter. Also, the `--scenario` option will be fully explained in a later section.

3.3 Graphical User Interface

A much more user-friendly way to use Taxi Simulator is through the built-in graphical user interface. This interface is accessed via any web browser. Just look at the address shown on the screen when you run the simulator and access that website.

Hint: The Coordinator agent is who raises the GUI and shows the address in the debug:

```
INFO:CoordinatorAgent:Web interface running at http://127.0.0.1:9000/app
```

This address is (in most cases): <http://127.0.0.1:9000/app>

Once you visit the GUI address you see an interface like this:

In the GUI you can see a map of the city on the right and a Control Panel with various options on the left:

1. Two selectors to set the number of taxis and passengers and an **Add** button. When this button is pressed the number of taxis and passengers that are in the input boxes are created in random positions inside the map. This form is very similar to the command line option, except that you can add Taxi and Passenger agents at any time during the simulation.
2. A **Run** button that starts the simulation.
3. Stats of the waiting time and total time of the simulation in real time.
4. A collapsable tree view with the taxis and passengers that are included in the simulation, with a color bullet that indicates their current status.

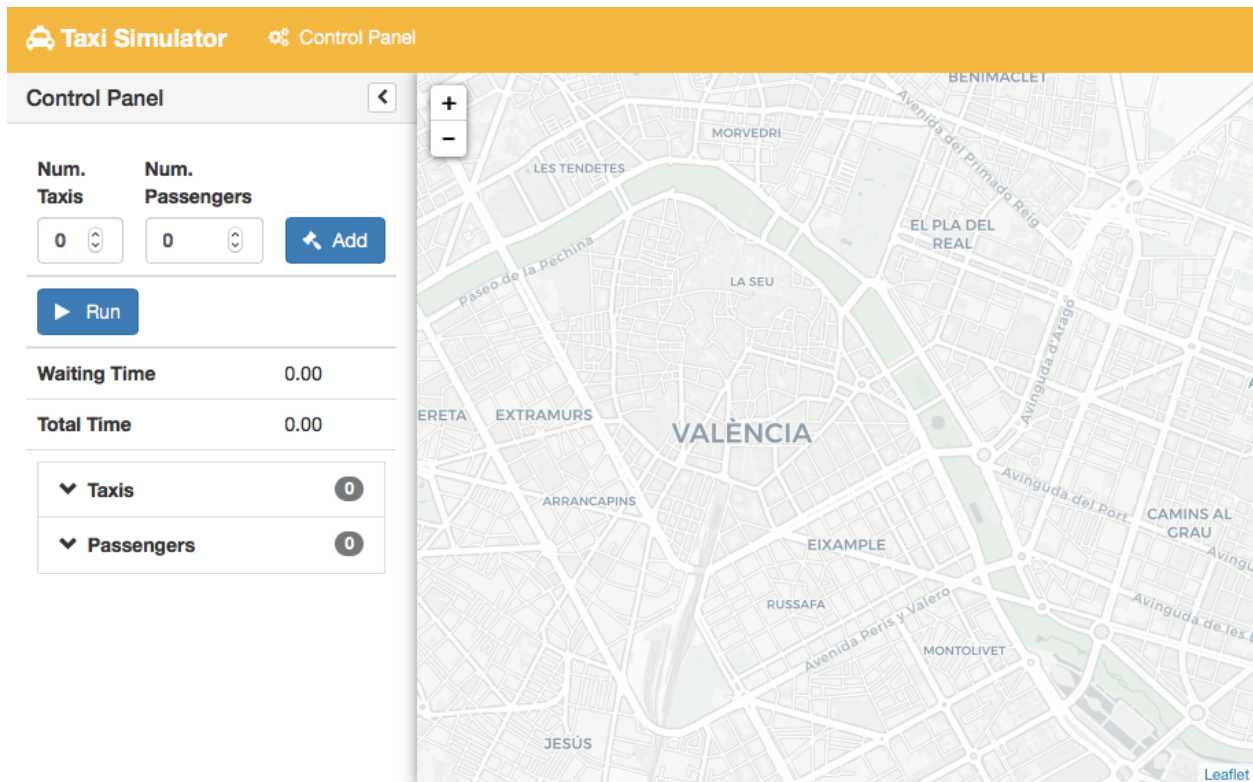










Fig. 3.1: GUI at startup

If the **Run** button is pressed the simulation shows how the taxis move to the passengers and deliver them to their destinations.

Notice that when a taxi picks up a passenger, the passenger's icon disappears from the map view (since it is inside the taxi) and is no longer viewed (it's also not shown when it arrives at its destination). However, you can check at any time your passengers status in the tree view of the Control Panel.

The code colors in the tree view indicate the status of a taxi or a passenger. The legend of colors is as follows:

Taxis		Passengers	
Bullet	Status	Bullet	Status
	WAITING		WAITING
	WAITING FOR APPROVAL		ASSIGNED
	MOVING TO PASSENGER		IN TAXI
	MOVING TO DESTINATION		IN DESTINATION

Hint: Every time than a bullet is pulsing means that the agent is moving.

When a taxi is moving it's also shown in the GUI the path that the taxi is following. The color of the path indicates the

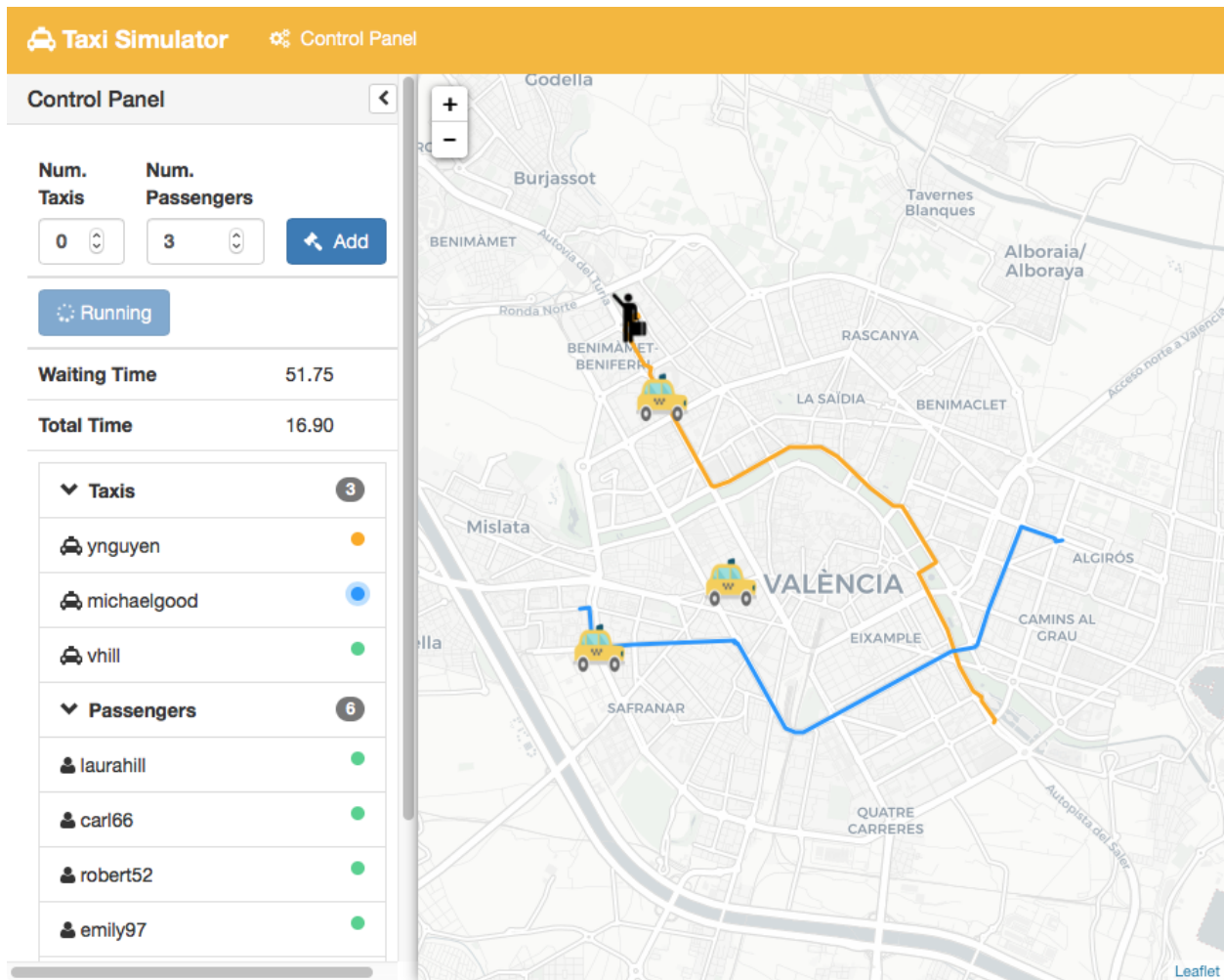


Fig. 3.2: Simulation in progress

type of movement than the taxi is doing. A yellow path indicates that the taxi is going to pick up the passenger. On the other hand, a blue path indicates that the taxi is taking the passenger to his destination.

Note: A simulation is finished when all taxis are free (and waiting for new passengers) and all passengers are in their destinations (i.e. all bullets are green).

3.4 Loading Scenarios

Adding agents using both the graphical interface and command line is convenient and fast, but if you want to perform repeatable experiments where you choose where agents appear and what the destinations of the passengers are (rather than random data) then you need the mechanism of the **scenarios**.

The ability to load scenarios to Taxi Simulator allows us to repeat the same experiment as many times as we want with the same initial conditions. Taxi Simulator supports to load a *scenario* file that defines all the fields that you need to load the same information repeatedly. A scenario file must be coded in JSON format.

The fields that the scenario file must include are a passengers list and a taxis list. Each passenger must include the following fields:

Field	Description
position	Initial coordinates of the passenger
dest	Destination coordinates of the passenger
name	Name of the passenger
password	Password for registering the passenger in the platform (optional)

For taxis the fields are as follows:

Field	Description
position	Initial coordinates of the taxi
name	Name of the taxi
password	Password for registering the taxi in the platform (optional)
speed	Speed of the taxi (in meters per second)

An example of a scenario file with two passengers and two taxis:

```
{
  "passengers": [
    {
      "dest": [ 39.463356, -0.376463 ],
      "position": [ 39.460568, -0.352529 ],
      "name": "michaelstewart",
      "password": "T3TnmjuI(m"
    },
    {
      "dest": [ 39.49529, -0.401478 ],
      "position": [ 39.49529, -0.401478 ],
      "name": "ghiggins",
      "password": "@5wPA$Mx#O"
    }
  ],
  "taxis": [
    {
```

```
    "position": [ 39.462618, -0.364888 ],
    "name": "taxi1",
    "password": "$JM!Zcwh0R",
    "speed": 2000
  },
  {
    "position": [ 39.478458, -0.406736 ],
    "password": "_bx1TBEiu8",
    "name": "taxi2",
    "speed": 2000
  }
]
```

Finally, to load a scenario in a simulation use the `--scenario` option with the filename of the JSON file:

```
$ taxi_simulator --scenario my_scenario.json

INFO:root:Starting Taxi Simulator
INFO:CoordinatorAgent:Coordinator agent running
INFO:CoordinatorAgent:Web interface running at http://127.0.0.1:9000/app
INFO:root:Creating 0 taxis and 0 passengers.
INFO:root:Loading scenario my_scenario.json
INFO:RouteAgent:Route agent running
```

Developing New Strategies

Table of Contents

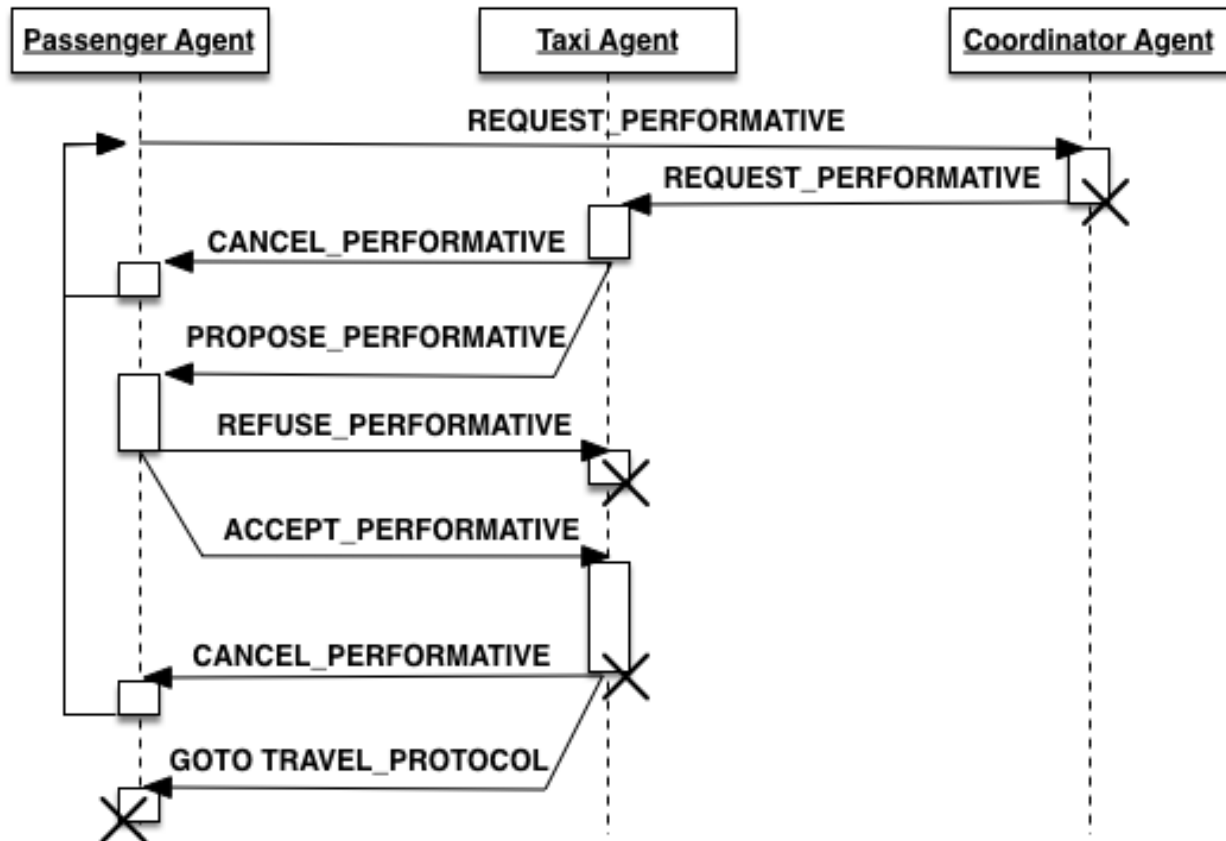
- *Developing New Strategies*
 - *Introduction*
 - * *Description of the Coordinator Agent*
 - *Strategy Behaviour (DelegateRequestTaxiBehaviour)*
 - * *Description of the Taxi Agents*
 - *Strategy Behaviour (AcceptAlwaysStrategyBehaviour)*
 - *Moving Behaviour*
 - * *Description of the Passenger Agents*
 - *Strategy Behaviour*
 - *Travel Behaviour*
 - * *The Negotiation Process between Taxi and Passenger Agents*
 - *Agent Foundations*
 - * *SPADE*
 - *Agent Model: Behaviors*
 - *Communication API, Messages and Templates*
 - *How to implement your own strategies*
 - * *The Strategy Pattern*
 - * *The Strategy Behaviour*
 - *Helpers*

- * *Developing the Coordinator Agent Strategy*
 - *Code*
 - *Helpers*
 - * *Developing the Taxi Agent Strategy*
 - *Code*
 - *Helpers*
 - * *Developing the Passenger Agent Strategy*
 - *Code*
 - *Helpers*
 - * *Other Helpers*
- *How to Implement New Strategies (Level 1) – Recommendations*

4.1 Introduction

One of the main features of “Taxi Simulator” is the ability to change the default negotiation strategy of the agents that interact during the simulation: the Coordinator agent, the Taxi agents and the Passenger agents. The overall goal of the negotiation strategy of these three agent types is to decide which Taxi agent will transport each Passenger agent to its destination, making sure that no Passenger agent is left unattended. Additionally, the negotiation strategy may also try to optimize some metrics, such as the average time that Passenger agents are waiting to be served, or that the amount of gas spent by Taxi in their movements.

The negotiation strategy is based on two main elements. First, it is based on the internal logic of each agent type (Coordinator, Taxi and Passenger) and, in particular, on their respective *strategy behavior*, which includes the internal logic of each agent type regarding the negotiation process. And second, it is also based on the so-called *REQUEST* protocol, which comprises the types of messages exchanged among the three agent types during the negotiation. The following diagram presents the protocol in the typical FIPA format, where agents types are depicted as vertical lines and the exchanged message types (or “performatives”) in horizontal arrows:



This chapter introduces first the current, default strategy of each agent type (Coordinator, Taxi and Passenger) and then explains how to introduce new strategies for any, or all, of them.

4.1.1 Description of the Coordinator Agent

The Coordinator Agent is responsible for putting in contact the Passenger agents that need a taxi service, and the Taxi agents that may be available to offer these services. In short, the Coordinator Agent acts like a taxi call center, accepting the incoming requests from customers (Passenger agents) and forwarding these requests to the (appropriate) Taxi agents. In order to do so, the Coordinator agent knows the names and addresses of every Passenger and Taxi agent registered in the system.

In the context of the Taxi Simulator, a “taxi service” involves, once a particular Passenger and Taxi agents have reached an agreement, the movement of the Taxi agent from its current position to the Passenger’s position in order to pick the Passenger up, and then the transportation of the Passenger agent to its destination.

The Coordinator Agent includes a single behavior, which is its strategy behavior, now described.

Strategy Behaviour (*DelegateRequestTaxiBehaviour*)

The goal of the strategy behavior of the Coordinator Agent is basically to **receive** the “request” messages (*REQUEST_PERFORMATIVE*) sent by the Passenger agents that need a taxi service and, for each request, selecting the Taxi agent, or agents, that may perform the service, and **forward** the request to them. A *REQUEST_PERFORMATIVE* message includes the following fields:

<pre>"passenger_id": Id of the Passenger agent that performs the request. "origin": Current position of the Passenger, where the Taxi has to pick it up. "dest": Destination of the Passenger, where the Taxi needs to transport it.</pre>

The particular set of Taxi agents to which the request will be forwarded depends on the *allocation policy* of the Coordinator Agent, which is part of the strategy. In the default strategy behavior for the Coordinator agent (*DelegateRequestTaxiBehaviour*), the allocation policy is the simplest possible: it forwards every incoming request to **all** the Taxi agents, regardless of their current statuses or any other consideration (such as, for example, the last time they performed a service, or the distance between them and the Passenger agent).

In the default strategy behavior, the set of incoming messages that may be delivered to the Coordinator Agent is reduced to the requests made by Passenger agents, and the behavior itself does not include multiple states. So, each incoming message is processed in the same way, and leaves the behavior in the same (unique) state.

Once each request has been forwarded to some (or all) the Taxi agents, the goal of the Coordinator Agent for that request is achieved. This is the starting point to the negotiation between the Passenger that has issued the request and the Taxi agents that have received it, which is described in the following sections.

4.1.2 Description of the Taxi Agents

The Taxi agents represent vehicles which can transport Passenger agents from their current positions to their respective destinations. In order to do that, Taxi agents incorporate two behaviors: the strategy behavior and the moving behavior, now described.

Strategy Behaviour (*AcceptAlwaysStrategyBehaviour*)

The goal of the strategy behavior of a Taxi agent is to negotiate with Passenger agents which are requesting a taxi service the conditions of the service offered by the Taxi, in order to achieve an agreement with these Passenger agents. When an agreement is reached between a particular Passenger and Taxi agents, then the Taxi agent picks up the Passenger agent and transport it to its destination (and starts the Moving Behavior, described below).

The currently implemented, default strategy behavior is called *AcceptAlwaysStrategyBehaviour*, and has a direct relation with the *REQUEST* protocol explained above. In particular, the behavior can be thought of as a finite-state machine with some different states specifying the statuses of the Taxi agent regarding the strategy behavior, and some transitions between states, which are triggered either by messages (of the *REQUEST* protocol) received by the Taxi agent, or by some other program conditions. This is depicted in the following diagram:

The semantics of each state are now described:

- **TAXI_WAITING**: In this state, the Taxi agent is available (free) and waiting for requests from Passenger agents. While in this state, if it receives a request message (*REQUEST_PERFORMATIVE*) from a particular Passenger agent, it will send the Passenger a service proposal (*PROPOSE_PERFORMATIVE*) and it will change its state to *TAXI_WAITING_FOR_APPROVAL*.
- **TAXI_WAITING_FOR_APPROVAL**: In this state, the Taxi agent is waiting for the response message from a Passenger agent to which it has sent a service proposal message. While in this state, it may receive two alternative answers from the Passenger agent: (1) the Passenger refuses the service proposal (*REFUSE_PERFORMATIVE*), in which case the Taxi changes its state back to *TAXI_WAITING*; or (2) the Passenger accepts the proposal (*ACCEPT_PERFORMATIVE*), in which case it will change to the state *TAXI_MOVING_TO_PASSENGER*.
- **TAXI_MOVING_TO_PASSENGER**: In this state, the Taxi agent and the Passenger agent have agreed to perform a taxi service, and then the Taxi agent starts to travel to the Passenger location in order to pick it up. This is the final state of the negotiation between the Taxi and a certain Passenger agent. In this state, the Taxi agent executes the helper function *pick_up_passenger*, which automatically starts the so-called Moving Behavior in the Taxi agent, described below. It also sends a message to the Travel Behavior of the Passenger agent, which starts that behavior (this is explained in the next section).

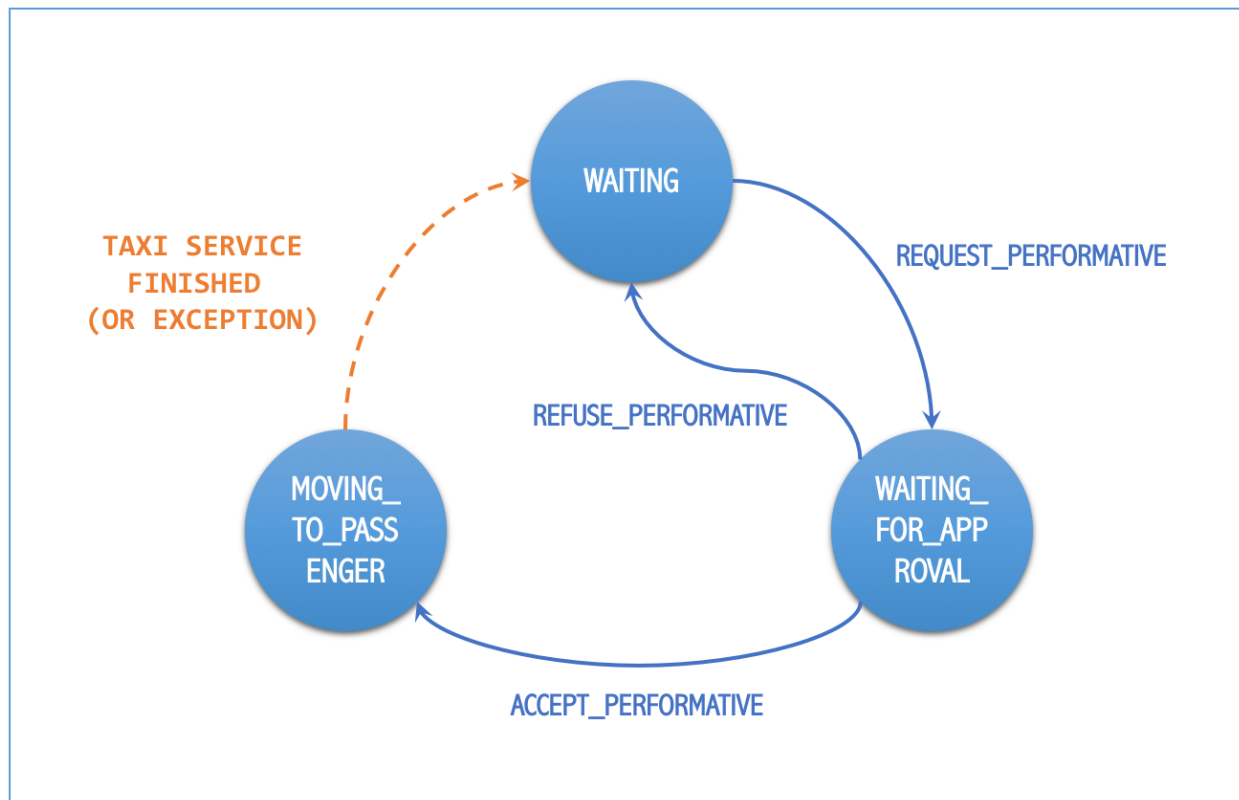


Fig. 4.1: States and transitions of the strategy behavior of a Taxi agent.

Moving Behaviour

This behavior makes the Taxi agent to move to the current location of the Passenger agent with which it has reached an agreement to perform a taxi service. After picking the Passenger agent up, the Taxi will then transport it to its destination. During that travel, the behavior informs the Passenger agent of where the Taxi is and what it is doing (going to pick up the Passenger, taking the Passenger to its destination, reaching the destination, etc.). All this is performed by sending the Passenger agent some messages which belong of another, dedicated protocol called *TRAVEL_PROTOCOL*.

Once the Taxi reaches the Passenger agent's destination and the Passenger agent is informed about it, the state of the Taxi agent (of the strategy behavior) is here changed to *TAXI_WAITING*, indicating that it is now free, and hence making the Taxi agent available again to receiving new requests from other Passenger agents.

Warning: This behavior is internal and automatic, and it is not intended to be modified while developing new negotiation strategies. The same applies to the *TRAVEL_PROTOCOL* protocol.

4.1.3 Description of the Passenger Agents

The Passenger agents represent people that need to go from one location of the city (their “current location”) to another (their “destination”), and for doing so, they request a taxi service. Each Passenger agent requires a single taxi service and so, once transported to its destination, it reaches its final state and ends its execution. During that execution, Passenger agents incorporate two behaviors: the strategy behavior and the travel behavior, now described.

Strategy Behaviour

In the course of the *REQUEST* protocol, the request of a taxi service made by a Passenger agent is answered by one (or several) Taxi agents, each of which offering the Passenger their conditions to perform such service. The goal of the strategy behavior of a Passenger agent is to select the best of these taxi service proposals, according to its needs and/or preferences (e.g., to be picked up faster, to get the nearest available taxi, to get the cheapest service, etc.).

The currently implemented, default strategy behavior is called *AcceptFirstRequestTaxiBehaviour*. As in the strategy behavior of the Taxi agents above, here we can also consider the strategy as a finite-state machine related to the messages (of the *REQUEST* protocol) received by the Passenger agent, as depicted below:

The semantics of each state are now described:

- *PASSENGER_WAITING*: In this state, the Passenger agent requires a taxi service and, periodically, sends a request for that service until one (or many) Taxi agent proposals (*PROPOSE_PERFORMATIVE*) are received. When the Passenger accepts a particular proposal (in the current implementation, always the first one it receives while in this state) then it communicates so to the proposing Taxi agent, and changes its own status to *PASSENGER_ASSIGNED*.
- *PASSENGER_ASSIGNED*: In this state, the Passenger agent has been assigned to a particular taxi, and the taxi service is being produced. The Passenger side of the taxi service is implemented by activating the Travel Behavior, described below, which is started by a message sent by the Taxi agent (in its helper function *pick_up_passenger*). If something goes wrong (for example, an exception is raised during the taxi service) or the Taxi agent voluntarily wants to cancel the service, then the Taxi agent sends a *CANCEL_PERFORMATIVE* to the Passenger agent, which would then change its status back to *PASSENGER_WAITING*, initiating the request process again.

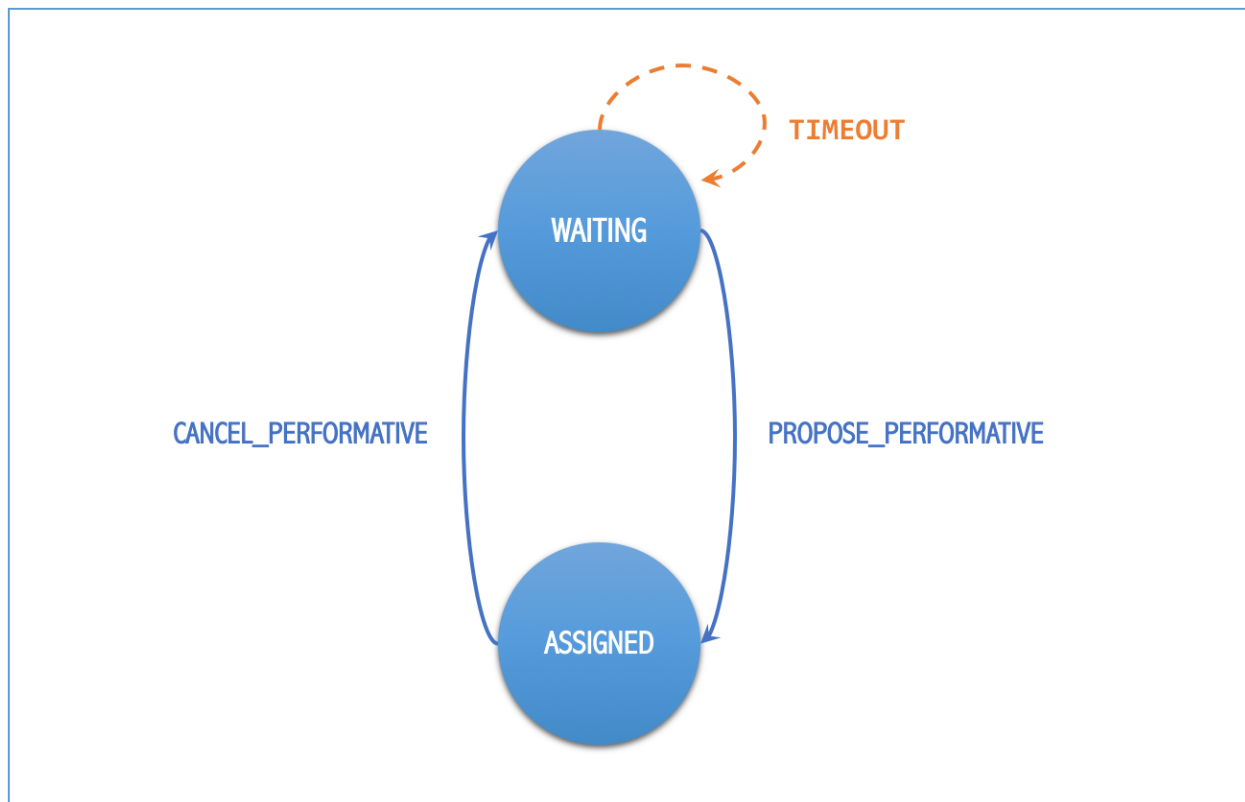


Fig. 4.2: States and transitions of the strategy behavior of a Passenger agent.

Travel Behaviour

This behavior is activated (in the Passenger agent) when a Taxi agent decides to pick up the Passenger agent, by means of a message sent by the Taxi (inside the Taxi agent's helper function *pick_up_passenger*). This message, as well as other messages sent by the Taxi agent to this behavior, belongs to a protocol called the *TRAVEL_PROTOCOL*.

The messages of the *TRAVEL_PROTOCOL* drive the transitions between the different states of this behavior, in the same way that the *REQUEST_PROTOCOL* does for the strategy behavior. In particular, the states of this behavior are: *PASSENGER_IN_TAXI*, when the Taxi agent has reached the Passenger agent's position and has picked it up; and *PASSENGER_IN_DEST*, when the Taxi agent has reached the Passenger agent's destination. This would be the final state of the Passenger agent.

Warning: This behavior is internal and automatic, and it is not intended to be modified while developing new negotiation strategies. The same applies to the *TRAVEL_PROTOCOL* protocol.

4.1.4 The Negotiation Process between Taxi and Passenger Agents

After separately explaining the strategy behavior of Taxi and Passenger agents, this section tries to relate both behaviors. This is important to understand how these two agent types interact with each other in order to coordinate and reach the overall goals of the simulation.

In particular, there are three key aspects (embedded within the strategy behaviors) which influence the overall coordination process implemented in the simulator, as now described:

- The conditions of a taxi service proposal. The current implementation does not consider any special condition other

than the Taxi agent being free (available to perform the service). Some aspects that could be included in a taxi proposal would be, for example, the current location of the taxi, the proposed fare, the route to take the Passenger agent to its destination, etc.

- The preferences of passengers in order to select a particular taxi proposal. In the current implementation, the

Passenger agents always accept the first proposal received from a Taxi agent. In a more sophisticated negotiation, some internal goals/conditions of the Passenger agent could be taken into account in order to select a "better" proposal. These might include, for example, the expected waiting time until the Taxi agent arrives, the amount of money that the service is expected to cost, the brand of the Taxi vehicle, etc.

- The possibility of a taxi to voluntarily cancel an ongoing taxi service after a proposal has been accepted by a passenger.

This may happen only before the passenger has been picked up, that is, while the taxi is moving from its initial position to the location where the passenger is waiting for it. In the current implementation, a taxi service cancellation can only be produced if some exception is raised while the service is being produced (for example, if the software calculating a route for the Taxi agent fails to produce a valid route). Since new Passenger (and maybe Taxi) agents can appear at any time while the simulation is running, a voluntary cancellation of taxi services could improve the overall transportation of passengers throughout the simulation, allowing for a "dynamic reallocation" of passengers to taxis, even when taxi services were already committed.

4.2 Agent Foundations

The architecture of Taxi Simulator is built on top of a multi-agent system platform called SPADE. Although it is not necessary to build new agents in order to develop new coordination strategies (the simulator provides all the necessary

agents), it is interesting to know how they work and what methods they provide for the creation of coordination strategies.

Next we will present the SPADE platform and its main features. For more documentation you can visit their website <https://github.com/javipalanca/spade>.

4.2.1 SPADE

SPADE (Smart Python multi-Agent Development Environment) is a multi-agent system (MAS) platform based on the *XMPP* technology and written in the *Python* programming language. This technology offers by itself many features and facilities that ease the construction of MAS, such as an existing communication channel, the concepts of users (agents) and servers (platforms) and an extensible communication protocol based on XML.

Extensible Messaging and Presence Protocol (XMPP) is an open, XML-inspired protocol for near-real-time, extensible instant messaging (IM) and presence information. The protocol is built to be open and free, asynchronous, decentralized, secure, extensible and flexible. These last two features allow XMPP not only to be an instant messaging protocol, but it can also be extended and used for many tasks and situations (IoT, WebRTC, social, ...). SPADE itself uses some XMPP extensions to provide extended features to its agents, like remote procedure calls between agents (Jabber-RPC), file transfer (In-Band Bytestreams), and so on.

In order to fully understand how SPADE works, it is necessary to know how the agents are made up and how they communicate. In the following sections we will see the SPADE agent model and its communication API.

Agent Model: Behaviors

SPADE agents are threaded-based objects that can be run concurrently and that are connected to a SPADE platform which internally runs an XMPP server. Each agent must provide an ID and password for its connection to the platform. This is called the JID and has the form of an email: a user name string plus a “@” character plus the IP of the SPADE server (e.g. *my_agent@127.0.0.1*).

The internal components of the SPADE agents that conduct their intelligence are the **Behaviors**. A behavior is a task that an agent can run using different repeating patterns. SPADE agents can run several behaviors simultaneously. The most basic behavior type is the cyclic behavior, which repeatedly executes the same method over and over again indefinitely. This is the way to develop behaviors that wait for a perception, reason about it and finally execute an action and wait again for the next perception.

The following example is a sample of an agent with a cyclic behavior (`spade.Behaviour.Behaviour` type) that waits for a perception of the keyboard input, reasons on it and executes an action, indefinitely until the user presses Ctrl+C. To build a behavior you must inherit from the type of behavior you want (in the case of this example the cyclic behaviour is implemented in the class `spade.Behaviour.Behaviour`) and overload the method `_process()` where the body of the behavior is implemented. If needed you can also overload the `onStart()` and `onEnd()` methods to perform actions on the initialization or shutdown of a behavior

```
import spade
import datetime
import time

class MyAgent(spade.Agent.Agent):
    class MyBehavior(spade.Behaviour.Behaviour):

        def onStart(self):
            print("Initialization of behavior")

        def _process(self):
            # wait for perception, raw_input is a blocking call
            perception = raw_input("What's your birthday year?")
```

```
# reason about the perception
age = datetime.datetime.now().year - perception
# execute an action
print("You are " + str(age) + " years old.")

def onEnd(self):
    print("Shutdown of behavior")

def _setup(self):
    # Create behavior
    behavior = self.MyBehavior()
    # Register behavior in agent
    self.addBehaviour(behavior)

if __name__ == "__main__":
    a = MyAgent(agentjid="agent@127.0.0.1", password="secret")
    a.start()
    while True:
        try:
            time.sleep(1)
        except KeyboardInterrupt:
            break
    a.stop()
```

There are also other types of behaviors like one-shot behaviors, periodic behaviors, finite-state machine behaviors, etc.

Communication API, Messages and Templates

Communication is one of the cornerstones of a multi-agent system and SPADE is no exception. Agents can send and receive messages using a simple API and even more, they can receive them in certain behaviors according to templates they can define.

A `spade.ACLMessage.ACLMessage` is the class that needs to be filled in order to send a message. It follows the *FIPA Agent Communication Language* specifications or [FIPA ACL](#). An `ACLMessage` may be filled with several information, but the most important fields are the receiver, the content, the performative and the protocol. The receiver must be filled with an `spade.AID.aid` object, which is an `AgentID`. The content is a string-based body of the message. The performative and protocol both add semantic information to the conversation. They are usually used to represent the action and the rules that determine how the agents are going to communicate in a specific semantic context.

Tip: It's usually recommended to use a representation language for the content of the message. There are semantic languages like OWL or RDF, but in the case of this simulator we use JSON representation for ease of use.

All these fields have a getter and setter function. An example is shown next:

```
import spade

receiver_aid = spade.AID.aid(name="receiver_agent@127.0.0.1",
                             addresses=["xmpp://receiver_agent@127.0.0.1"])
msg = spade.ACLMessage.ACLMessage()
msg.addReceiver(receiver_aid) # a message may be sent to multiple receivers
msg.setPerformative("request")
msg.setProtocol("my_custom_protocol")
msg.setBody("{\"a_key\": 'a_value'}")
```

Hint: Other fields that can be filled in the message are the content language (`setLanguage()`), the ontology (`setOntology()`) and so on.

The next step is to send the message. This is done with the `send()` method provided by a Behaviour. See an example:

```
import spade

class SenderAgent(spade.Agent.Agent):
    class SendBehav(spade.Behaviour.OneShotBehaviour):

        def _process(self):
            receiver = spade.AID.aid(name="receiver@127.0.0.1",
                                     addresses=["xmpp://127.0.0.1"])

            msg = spade.ACLMessage.ACLMessage()
            msg.setPerformative("inform")
            msg.setOntology("myOntology")
            msg.setLanguage("OWL-S")
            msg.addReceiver(receiver)
            msg.setContent("Hello World")

            self.send(msg) # send the message

    def _setup(self):
        print "MyAgent starting..."
        behav = self.SendBehav()
        self.addBehaviour(behav)
```

Since only behaviours can receive messages SPADE provides a mechanism to configure which behavior must receive each type of message. This is done with *ACLTemplates*. When an agent receives a new message it checks if the message matches each of the behaviors using a template with which they were registered. If there is a match, the message is delivered to the mailbox of the corresponding behavior and will be read when the behavior executes the `receive()` method. Otherwise, the message will be delivered to a default behaviour if it was registered (the default behavior is registered with the `setDefaultBehaviour()` method instead of `addBehaviour()`).

Note: The `receive()` method accepts an optional parameter: **timeout=seconds** to be a blocking method until the specified number of seconds has elapsed. If timeout is reached without a message, then `None` is returned. If timeout is 0, then the `receive()` function is non-blocking and returns a `spade.ACLMessage.ACLMessage` or `None`.

An `spade.Behaviour.ACLTemplate` is created using the same API of `spade.Behaviour.ACLMessage`:

```
import spade
template = spade.Behaviour.ACLTemplate()
template.setOntology("myOntology")
```

ACLTemplates must be wrapped with the `spade.Behavior.MessageTemplate` to be registered with a behavior.

Note: A `spade.Behavior.MessageTemplate` accepts boolean operators to combine *ACLTemplates* (e.g. `my_tpl = MessageTemplate(template1 & template2)`)

At this point we can already see how to build an agent that registers a behavior with a template and receives messages

that match that template:

```
import spade
import time

class RecvAgent(spade.Agent.Agent):
    class ReceiveBehav(spade.Behaviour.Behaviour):

        def _process(self):
            msg = self.receive(timeout=10)

            # Check whether the message arrived
            if msg is not None:
                assert "myOntology" == msg.getOntology()
                print("I got a message with the ontology 'myOntology'")
            else:
                print("I waited 10 seconds but got no message")
                time.sleep(1)

    def _setup(self):
        recv_behav = self.ReceiveBehav()
        template = spade.Behaviour.ACLTemplate()
        template.setOntology("myOntology")
        msg_tplt = spade.Behaviour.MessageTemplate(template)

        self.addBehaviour(recv_behav, msg_tplt)
```

These are the basics of SPADE programming. To use *Taxi Simulator* you would not need to create all these structures, templates and classes. But it is always better to know the foundations before we get down to business.

4.3 How to implement your own strategies

Taxi simulator is designed to allow students to implement and test new strategies that lead to system optimization. The goal of this educational simulator is to make it easier for students to work with new coordination strategies without going down to the mud. With this purpose, Taxi Simulator implements the Strategy design pattern, which allows students to test new coordination strategies without having to make major modifications in the application.

4.3.1 The Strategy Pattern

The **Strategy pattern** is a design pattern that enables selecting an algorithm at runtime. When in an application we have to implement different versions of an algorithm and we want to select at runtime a specific version of the algorithm, then the Strategy Pattern is the best choice for that purpose. With this pattern you can define a separate strategy in an object that encapsulates the algorithm. The application that executes the algorithm **must** define an interface that every implementation of the strategy will follow, as can be viewed in next figure:

Following this implementation the context object can call the current strategy implementation without knowing how the algorithm was implemented. This design pattern was created among others by a group of authors commonly known as the **Gang of Four** (E. Gamma, R. Helm, R. Johnson and J. Vlissides) and is well presented in [\[GangOfFour95\]](#).

Taxi Simulator uses the *Strategy Pattern* to allow students to implement three different strategies (one for the coordinator agent, one for the taxi agent and one for the passenger agent) without having to develop new agents or entering in the complexity of the simulator. Thanks to this pattern students can develop their strategies in an external file and pass it as an argument when the simulator is run.

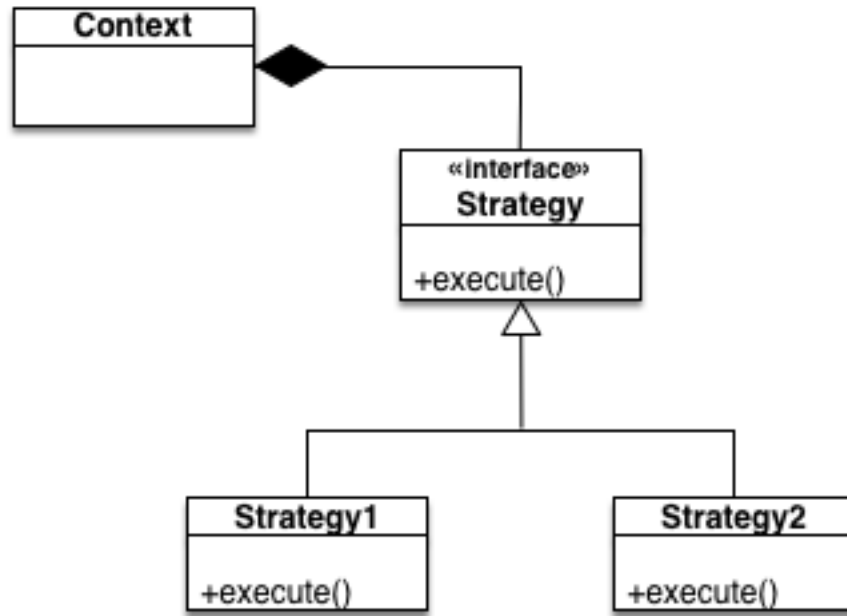


Fig. 4.3: The Strategy Pattern UML

Taxi Simulator implements three interfaces for these agents and each interface provides also some helper functions that intend to make easier some common actions that each subclassed agent usually has to do. These three interfaces inherit from the `StrategyBehaviour` class and are called: `CoordinatorStrategyBehaviour`, `TaxiStrategyBehaviour` and `PassengerStrategyBehaviour`.

4.3.2 The Strategy Behaviour

The `StrategyBehaviour` is the metaclass from which interfaces are created for the strategies of each agent in the simulator. It inherits from a `spade.Behaviour.Behaviour` class, so when implementing it you will have to overload the `_process()` method that will run cyclically endlessly until the agent stops.

Helpers

The Strategy Behaviour provides also some helper functions that are widely useful for any kind of agent in the simulator. We have already read about the `send()` and `receive()` functions, that allow agents to communicate with each other. The rest of the helper functions allow to store and retrieve information in the agent and to log messages.

```

def receive(self, timeout=5)
def send(self, message)

def store_value(self, key, value)
def get_value(self, key)
def has_value(self, key)
  
```

Danger: Don't store information in the Behaviour itself since it is a cyclic behaviour and is run by calling repeatedly the `_process()` function, so the context of the function is not persisted.

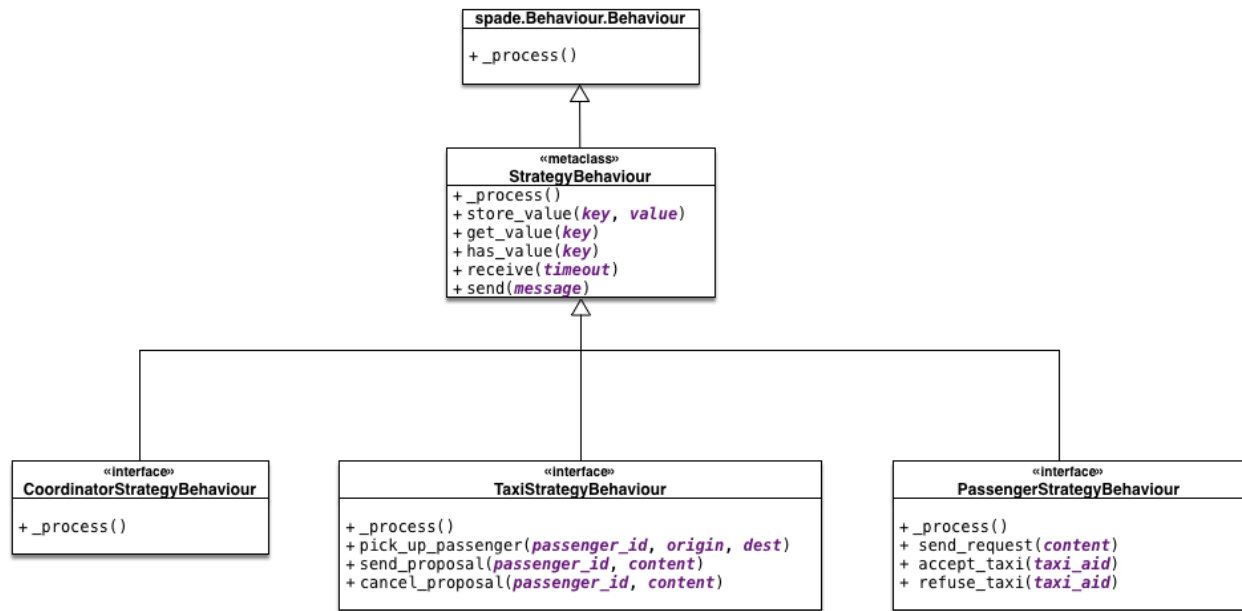


Fig. 4.4: The StrategyBehaviour class and their inherited interfaces

The `store_value()`, `get_value()` and `has_value()` functions allow to store persistent information in the agent and to recover it at any moment. The store uses a *key-value* interface to store your data.

There is also a very useful helper function which is the **logger**. This is not really a function but a system of logs which can be used to generate debug information at different levels. There are four levels of logging which are, in order of importance, the following:

- **DEBUG** Used with `self.logger.debug("my debug message")`. These messages are only shown when the simulator is called with the `-v` option. This is usually superfluous information.
- **INFO** Used with `self.logger.info("my info message")`. These messages are always shown and are the regular information shown in logs.
- **WARNING** Used with `self.logger.warn("my warning message")`. These messages are always shown and are used to show warnings to the user.
- **ERROR** Used with `self.logger.error("my error message")`. These messages are always shown and are used to show errors to the user.

4.3.3 Developing the Coordinator Agent Strategy

To develop a new strategy for the Coordinator Agent you need to create a class that inherits `CoordinatorStrategyBehaviour`. Since this is a cyclic behaviour class that follows the *Strategy Pattern* and that inherits from the `StrategyBehaviour`, it has all the previously presented helper functions for communication and storing data inside the agent.

Following the *REQUEST* protocol, the Coordinator agent is supposed to receive every request for a taxi from passengers and to carry out the action that your strategy determines (remember that in the default strategy `DelegateRequestTaxiBehaviour` the coordinator delegates the decision to all the taxis by redirecting all requests to all taxis without any previous or further reasoning).

The place in the code where your coordinator strategy must be coded is the `_process()` function. This function is executed in an infinite loop until the agent stops. In addition, you may overload also the `onStart()` and the

`onEnd()` functions to execute code before the creation of the strategy or after its destruction.

Code

As an example, this is the code of the default coordinator strategy `DelegateRequestTaxiBehaviour`:

```
from taxi_simulator.coordinator import CoordinatorStrategyBehaviour
from taxi_simulator.helpers import coordinator_aid

class DelegateRequestTaxiBehaviour(CoordinatorStrategyBehaviour):
    def _process(self):
        msg = self.receive(timeout=60)
        if msg:
            msg.removeReceiver(coordinator_aid)
            for taxi in self.get_taxi_agents():
                msg.addReceiver(taxi.getAID())
                self.logger.debug("Coordinator sent request to taxi {}".format(taxi.
↪getName()))
            self.send(msg)
```

Helpers

To make it easier for the student, the coordinator agent has two helper functions that allow her to recover a list of all the taxi agents and passenger agents registered in the system. These functions are:

- `get_taxi_agents()`
Returns a list of the taxi agents.
- `get_passenger_agents()`
Returns a list of the passenger agents.

4.3.4 Developing the Taxi Agent Strategy

To develop a new strategy for the Taxi Agent you need to create a class that inherits `TaxiStrategyBehaviour`. Since this is a cyclic behaviour class that follows the *Strategy Pattern* and that inherits from the `StrategyBehaviour`, it has all the previously presented helper functions for communication and storing data inside the agent.

The taxi strategy is intended to receive requests from passengers, forwarded by the coordinator agent, and to send proposals to that passengers in order to be selected by the corresponding passenger. If the taxi proposal is accepted, then it begins the process of going to the passenger's place, picking her up and taking her to the requested destination.

Warning: The process that implies a taxi movement is out of the scope of the strategy and should not be addressed by the strategy implementation. This passenger transfer process is automatically triggered when the strategy executes the helper function `pick_up_passenger()` (which is supposed to be the last action of a taxi strategy).

The place in the code where your taxi strategy must be coded is the `_process()` function. This function is executed in an infinite loop until the agent stops. In addition, you may overload also the `onStart()` and the `onEnd()` functions to execute code before the creation of the strategy or after its destruction.

Code

The default strategy of a taxi is to accept every passenger's requests if the taxi is not assigned to any other passenger or waiting a confirmation from any passenger. As an example, this is the code of the default taxi strategy `AcceptAlwaysStrategyBehaviour`:

```
from taxi_simulator.taxi import TaxiStrategyBehaviour

class AcceptAlwaysStrategyBehaviour(TaxiStrategyBehaviour):
    def _process(self):
        # wait for a message
        msg = self.receive(timeout=60)
        if not msg:
            # return if no new message
            return

        content = content_to_json(msg) # deserialize string content to JSON
        performative = msg.getPerformative()

        self.logger.debug("Taxi {} received request protocol from passenger {}".format(self.myAgent.agent_id, content["passenger_id"]))
        # a new request from a passenger has arrived
        if performative == REQUEST_PERFORMATIVE:
            if self.myAgent.status == TAXI_WAITING:
                # send a proposal with an empty content and wait for approval
                self.send_proposal(content["passenger_id"], {})
                self.myAgent.status = TAXI_WAITING_FOR_APPROVAL

            # my proposal has been accepted (Hooray!)
            elif performative == ACCEPT_PERFORMATIVE:
                # I should only receive an ACCEPT if I was waiting for it
                if self.myAgent.status == TAXI_WAITING_FOR_APPROVAL:
                    self.logger.debug("Taxi {} got accept from {}".format(self.myAgent.agent_id, content["passenger_id"]
↪"))

                    try:
                        # Change my status to MOVING and trigger pick_up_passenger.
↪Strategy is done.
                        self.myAgent.status = TAXI_MOVING_TO_PASSENGER
                        self.pick_up_passenger(content["passenger_id"], content["origin"],
↪ content["dest"])

                    except PathRequestException:
                        # If taxi is not able to get a path to the passenger, then it is
↪forced to cancel
                        self.logger.error("Taxi {} could not get a path to passenger {}".format(self.myAgent.getName(), content[
↪"passenger_id"]))
                        self.myAgent.status = TAXI_WAITING
                        self.cancel_proposal(content["passenger_id"])

                    except Exception as e:
                        self.logger.error("Unexpected error in taxi {name}: {exception}"
↪.format(name=self.myAgent.getName(),
↪exception=e))

                        self.cancel_proposal(content["passenger_id"])
                        self.myAgent.status = TAXI_WAITING
```

```

        else: # If I was not waiting for an ACCEPT then cancel proposal with the
        ↪passenger
            self.cancel_proposal(content["passenger_id"])

        # my proposal has been refused. Don't worry, return to WAITING status and get
        ↪over it.
        elif performative == REFUSE_PERFORMATIVE:
            self.logger.debug("Taxi {} got refusal from {}".format(self.myAgent.agent_
            ↪id,
                                                                    content["passenger_
            ↪id"]))
            self.myAgent.status = TAXI_WAITING

```

Helpers

In the example below there are some helper functions that are specific for the taxi strategy. These are:

```

def send_proposal(self, passenger_id, content=None)
def cancel_proposal(self, passenger_id, content=None)
def pick_up_passenger(self, passenger_id, origin, dest)

```

Let's present each one of them.

- `send_proposal()`

This helper function simplifies the composition and sending of a message to a passenger with a proposal. It sends an `ACLMessage` to `passenger_id` using the **REQUEST_PROTOCOL** and a **PROPOSE_PERFORMATIVE**. It optionally accepts a *content* parameter where you can include any information you may want the receiver to analyze.

- `cancel_proposal()`

This helper function simplifies the composition and sending of a message to a passenger to cancel a proposal. It sends an `ACLMessage` to `passenger_id` using the **REQUEST_PROTOCOL** and a **CANCEL_PERFORMATIVE**. It optionally accepts a *content* parameter where you can include any information you may want the receiver to analyze.

- `pick_up_passenger()`

This helper function triggers the **TRAVEL_PROTOCOL** of a taxi, which is the protocol that is used to transfer a passenger from its origin to its destination. This is an important function since it is usually the last action that a taxi strategy does, since from this point an alternative behaviour of the agent to transport the passenger begins and the strategy has finished its purpose (until the taxi is free again and receives a new request from a new passenger).

The `pick_up_passenger()` helper function receives as parameters the id of the passenger and the coordinates of the passenger's current position (*origin*) and its destination (*dest*).

4.3.5 Developing the Passenger Agent Strategy

To develop a new strategy for the Passenger Agent you need to create a class that inherits `PassengerStrategyBehaviour`. Since this is a cyclic behaviour class that follows the *Strategy Pattern* and that inherits from the `StrategyBehaviour`, it has all the previously presented helper functions for communication and storing data inside the agent.

The passenger strategy is intended to ask for a taxi to the coordinator agent, then wait for taxi proposals and, after evaluating them, choosing a taxi proposal to be taken to her destination.

The place in the code where your passenger strategy must be coded is the `_process()` function. This function is executed in an infinite loop until the agent stops. In addition, you may overload also the `onStart()` and the `onEnd()` functions to execute code before the creation of the strategy or after its destruction.

Code

The default strategy of a Passenger agent is a dummy strategy that accepts the first proposal it receives. As an example, this is the code of the default passenger strategy `AcceptFirstRequestTaxiBehaviour`:

```
from taxi_simulator.passenger import PassengerStrategyBehaviour

class AcceptFirstRequestTaxiBehaviour(PassengerStrategyBehaviour):
    def _process(self):
        # If I'm waiting then send a new request
        if self.myAgent.status == PASSENGER_WAITING:
            self.send_request(content={})

        # wait 5 seconds for a proposal
        msg = self.timeout_receive(timeout=5)

        if msg:
            performative = msg.getPerformative()
            taxi_aid = msg.getSender()

            # If I got a proposal then I blindly accept it
            if performative == PROPOSE_PERFORMATIVE:
                # But I accept it only if I was waiting for a proposal
                if self.myAgent.status == PASSENGER_WAITING:
                    self.logger.debug("Passenger {} received proposal from taxi {}".format(self.myAgent.agent_id, taxi_aid))

                    self.accept_taxi(taxi_aid)
                    self.myAgent.status = PASSENGER_ASSIGNED
                else:
                    # Otherwise I refuse the proposal (since I wasn't waiting for it)
                    self.refuse_taxi(taxi_aid)

            # If I receive a CANCEL performative it means my taxi has given up and I
            # 'm waiting again
            elif performative == CANCEL_PERFORMATIVE:
                if self.myAgent.taxi_assigned == taxi_aid.getName():
                    self.logger.warn("Passenger {} received a CANCEL performative_
from Taxi {}".format(self.myAgent.agent_id, taxi_aid))

                    self.myAgent.status = PASSENGER_WAITING
```

Helpers

In the example below there are some helper functions that are specific for the passenger strategy. These are:

```
def send_request(self, content=None)
def accept_taxi(self, taxi_aid)
def refuse_taxi(self, taxi_aid)
```

Let's present each one of them.

- `send_request()`

This helper is useful to make a new request without building the whole message (the helper functions makes it for you). It creates an *ACLMessage* with a **REQUEST** performative and sends it to the coordinator agent. In addition you can append a content to the request message to be used by the coordinator agent or the taxi agents (e.g. your origin coordinates or your destination coordinates).

- `accept_taxi()`

This is a helper function to quickly send an acceptance message to a `taxi_id`. It sends an *ACLMessage* with an **ACCEPT** performative to the selected taxi.

- `refuse_taxi()`

This is a helper function to quickly refuse a proposal from a `taxi_id`. It sends an *ACLMessage* with an **REFUSE** performative to the taxi whose proposal is being refused.

4.3.6 Other Helpers

Taxi Simulator comes also with a `helpers` module to provide some transversal support methods that may be useful for any agent. In this section we are showing each one of them.

- `build_aid()`

This function helps to create an `spade.AID.aid` object using the name of an agent as a parameter. This helps to create a structure that is very used when working with spade agents. It accepts a string with the name of the agent (e.g. “coordinator”) and returns a `spade.AID.aid` instance to be used in a `spade.ACLMessage.ACLMessage`.

Example:

```
taxi_aid = build_aid("taxi_1234")

assert taxi_aid.getName() == "taxi_1234@127.0.0.1"
assert taxi_aid.getAddresses() == ["xmpp://taxi_1234@127.0.0.1"]
```

- `coordinator_aid`

Since the coordinator agent is a very common agent and needed by almost every passenger and taxi agent, the `helpers` module provides a static `spade.AID.aid` instance to communicate with the coordinator agent.

- `content_to_json()`

Taxi Simulator uses the [JSON](#) data-interchange format to use as the content language of the messages (however, you can use the language you want, like RDF, XML, OWL, etc.). To facilitate the use of the JSON format we provide this helper function that receives a `spade.ACLMessage.ACLMessage` and returns the content of the message if JSON format (which is actually a `dict` object in Python).

Example:

```
msg = self.receive()

assert msg.getContent() == '{"my_coords": [39.253, -0.341]}'
assert content_to_json(msg) == {"my_coords": [39.253, -0.341]}
```

- `random_position()`

This helper function returns a random position in the map for being used if you need to create a new coordinate.

Example:

```
assert random_position() == [39.253, -0.341]
```

- `are_close()`

This helper function facilitates working with distances in maps. This helper function accepts two coordinates (`coord1` and `coord2`) and an optional parameter to set the tolerance in meters. It returns `True` if both coordinates are closer than the tolerance in meters (10 meters by default). Otherwise it returns `False`.

Example:

```
assert are_close([39.253, -0.341], [39.351, -0.333], 1000) == True
```

- `distance_in_meters()`

This helper function returns the distance in meters between two points.

Example:

```
assert distance_in_meters([-0.37565, 39.44447], [-0.40392, 39.45293]) == ↪  
↪3264.7134341427977
```

4.4 How to Implement New Strategies (Level 1) – Recommendations

At this point is time for you to implement your own strategies to optimize the problem of dispatching taxis to passengers. In this chapter we have shown you the tools to create these strategies. You have to create a file (in this example we are using `my_strategy_file.py`) and develop the strategies to be tested following the next template:

```
from taxi_simulator.coordinator import CoordinatorStrategyBehaviour
from taxi_simulator.passenger import PassengerStrategyBehaviour
from taxi_simulator.taxi import TaxiStrategyBehaviour

#####
#
#                               Coordinator Strategy
#
#####
class MyCoordinatorStrategy(CoordinatorStrategyBehaviour):
    def _process(self):
        # Your code here

#####
#
#                               Taxi Strategy
#
#####
class MyTaxiStrategy(TaxiStrategyBehaviour):
    def _process(self):
        # Your code here

#####
#
#                               Passenger Strategy
#
#####
```



```
class MyPassengerStrategy (PassengerStrategyBehaviour) :  
    def _process (self) :  
        # Your code here
```

In this file, three strategies have been created for the three types of agent handled by the simulator. We have called these strategies `MyCoordinatorStrategy`, `MyTaxiStrategy` and `MyPassengerStrategy`.

To run the simulator with your new strategies the command line interface accepts three parameters with the name of the file (without extension) and the name of the class of each strategy.

```
$ taxi_simulator --taxi my_strategy_file.MyTaxiStrategy  
                 --passenger my_strategy_file.MyPassengerStrategy  
                 --coordinator my_strategy_file.MyCoordinatorStrategy
```

Warning: The file must be in the current working directory and it must be referenced *without* the extension (if the file is named `my_strategy_file.py` use `my_strategy_file` when calling the simulator).

Once run the simulator you can test your strategies using the graphical web interface or by inspecting the output of the logs in the command line.

Information on specific functions, classes, and methods.

5.1 taxi_simulator package

5.1.1 Submodules

5.1.2 taxi_simulator.cli module

Console script for taxi_simulator.

5.1.3 taxi_simulator.coordinator module

```
class taxi_simulator.coordinator.CoordinatorAgent (agentjid, password, debug,  
                                                    http_port, backend_port, de-  
                                                    bug_level)
```

Bases: spade.Agent.Agent

Coordinator agent that manages the requests between taxis and passengers

add_passenger (*agent*)

Adds a new PassengerAgent to the store.

Parameters **agent** (PassengerAgent) – the instance of the PassengerAgent to be added

add_strategy (*strategy_class*)

Injects the strategy by instantiating the *strategy_class*. Since the *strategy_class* inherits from `spade.Behaviour.Behaviour`, the new strategy is added as a behaviour to the agent.

Parameters **strategy_class** (*class*) – the class to be instantiated.

add_taxi (*agent*)

Adds a new TaxiAgent to the store.

Parameters `agent` (TaxiAgent) – the instance of the TaxiAgent to be added

`clean_controller()`

Web controller that resets the simulator to a clean state.

Returns no template is returned since this is an AJAX controller, a dict with status=done

Return type None, dict

`entities_controller()`

Web controller that returns a dict with the entities of the simulator and their statuses.

Example of the entities returned data:

```
{
  "passengers": [
    {
      "status": 24,
      "taxi": "taxi2@127.0.0.1",
      "dest": [ 39.463356, -0.376463 ],
      "waiting": 3.25,
      "position": [ 39.460568, -0.352529 ],
      "id": "michaelstewart"
    }
  ],
  "taxis": [
    {
      "status": 11,
      "passenger": "michaelstewart@127.0.0.1",
      "assignments": 1,
      "path": [
        [ 39.478328, -0.406712 ],
        [ 39.478317, -0.406814 ],
        [ 39.460568, -0.352529 ]
      ],
      "dest": [ 39.460568, -0.352529 ],
      "position": [ 39.468131, -0.39685 ],
      "speed": 327.58,
      "id": "taxi2",
      "distance": "6754.60"
    }
  ],
  "stats": {
    "totaltime": "-1.00",
    "waiting": "3.25",
    "finished": False,
    "is_running": True
  },
  "tree": {
    "name": "Agents",
    "children": [
      {
        "count": "1",
        "name": "Taxis",
        "children": [ { "status": 11, "name": " taxi2", "icon": "fa-
↩taxi" } ]
      },
      {
        "count": "1",
        "name": "Passengers",
```

```

        "children": [ { "status": 24, "name": " michaelstewart", "icon
↪": "fa-user" } ]
        }
    ]
},
    "authenticated": False
}

```

Returns no template is returned since this is an AJAX controller, a dict with the list of taxis, the list of passengers, the tree view to be showed in the sidebar and the stats of the simulation.

Return type None, dict

generate_tree()

Generates the tree view in JSON format to be showed in the sidebar.

Returns a dict with all the agents in the simulator, with their name, status and icon.

Return type dict

get_passenger_stats()

Creates a dataframe with the simulation stats of the passengers The dataframe includes for each passenger its name, waiting time, total time and status.

Returns the dataframe with the passengers stats.

Return type pandas.DataFrame

get_simulation_time()

Returns the elapsed simulation time to the current time. If the simulation is not started it returns 0.

Returns the whole simulation time.

Return type float

get_stats()

Generates the stats of the simulation in JSON format.

Examples:

```

{
    "totaltime": "12.25",
    "waiting": "3.25",
    "finished": False,
    "is_running": True
}

```

Returns a dict with the total time, waiting time, is_running and finished values

Return type dict

get_taxi_stats()

Creates a dataframe with the simulation stats of the taxis The dataframe includes for each taxi its name, assignments, traveled distance and status.

Returns the dataframe with the taxis stats.

Return type pandas.DataFrame

get_value(key)

Returns a stored value from the agent's knowledge base.

Parameters **key** (*str*) – the name of the value

Returns The object stored with the key

Return type *object*

Raises *KeyError* – if the key is not in the knowledge base

has_value (*key*)

Checks if a key is registered in the agent's knowledge base

Parameters **key** (*str*) – the name of the value to be checked

Returns whether the knowledge base has or not the key

Return type *bool*

index_controller ()

Web controller that returns the index page of the simulator.

Returns the name of the template, the data to be pre-processed in the template

Return type *str, dict*

is_simulation_finished ()

Checks whether the simulation has finished or not. A simulation is finished if all passengers are at their destinations.

Returns whether the simulation has finished or not.

Return type *bool*

passenger_agents

Gets the list of registered passengers

Returns a list of *PassengerAgent*

Return type *list*

request_path (*origin, destination*)

Requests a path to the *RouteAgent*.

Parameters

- **origin** (*list*) – the origin coordinates (lon, lat)
- **destination** (*list*) – the target coordinates (lon, lat)

Returns the path as a list of points, the distance of the path, the estimated duration of the path

Return type *list, float, float*

run_controller ()

Web controller that starts the simulator.

Returns no template is returned since this is an AJAX controller, an empty data dict is returned

Return type *None, dict*

run_simulation ()

Starts the simulation

set_strategies (*coordinator_strategy, taxi_strategy, passenger_strategy*)

Gets the strategy strings and loads their classes. These strategies are prepared to be injected into any new taxi or passenger agent.

Parameters

- **coordinator_strategy** (*str*) – the path to the coordinator strategy
- **taxi_strategy** (*str*) – the path to the taxi strategy
- **passenger_strategy** (*str*) – the path to the passenger strategy

stop_agents ()

Stops the simulator and all the agents

store_value (*key*, *value*)

Stores a value (named by a key) in the agent's knowledge base that runs the behaviour. This allows the strategy to have persistent values between loops.

Parameters

- **key** (*str*) – the name of the value.
- **value** (*object*) – The object to be stored.

taxi_agents

Gets the list of registered taxis

Returns a list of TaxiAgent

Return type list

class taxi_simulator.coordinator.CoordinatorStrategyBehaviour

Bases: *taxi_simulator.utils.StrategyBehaviour*

Class from which to inherit to create a coordinator strategy. You must overload the `_process()` method

Helper functions:

- *get_taxi_agents()*
- *get_passenger_agents()*

get_passenger_agents ()

Gets the list of registered passengers

Returns a list of PassengerAgent

Return type list

get_taxi_agents ()

Gets the list of registered taxis

Returns a list of TaxiAgent

Return type list

onStart ()

5.1.4 taxi_simulator.helpers module

Helpers module

These functions are useful for the develop of new strategies.

exception taxi_simulator.helpers.AlreadyInDestination

Bases: *exceptions.Exception*

This exception is raised when an agent wants to move to a destination where it is already there.

exception `taxi_simulator.helpers.PathRequestException`

Bases: `exceptions.Exception`

This exception is raised when a path could not be computed.

`taxi_simulator.helpers.are_close(coord1, coord2, tolerance=10)`

Checks wheter two points are close or not. The tolerance is expressed in meters.

Parameters

- **coord1** (*list*) – a coordinate (longitude, latitude)
- **coord2** (*list*) – another coordinate (longitude, latitude)
- **tolerance** (*int*) – tolerance in meters

Returns whether the two coordinates are closer than tolerance or not

Return type `bool`

`taxi_simulator.helpers.build_aid(agent_id)`

Creates a new spade.AID.aid from a user string. :param agent_id: the name of the agent :type agent_id: str

Returns an Agent ID representing the agent.

Return type `spade.AID.aid`

`taxi_simulator.helpers.content_to_json(msg)`

Safely convert the content of a `spade.ACLMessage.ACLMessage` to a JSON format (dict). The content of the message MUST be in a string representation of the JSON format.

Parameters **msg** (`spade.ACLMessage.ACLMessage`) – an ACL message

Returns the content of the message loaded in a dict.

Return type `dict`

Raises `ValueError` – if no JSON object could be decoded

`taxi_simulator.helpers.distance_in_meters(coord1, coord2)`

Returns the distance between two coordinates in meters.

Parameters

- **coord1** (*list*) – a coordinate (longitude, latitude)
- **coord2** – another coordinate (longitude, latitude)

Returns distance meters between the two coordinates

Return type `float`

`taxi_simulator.helpers.kmh_to_ms(speed_in_kmh)`

Convert kilometers/hour to meters/second.

Parameters **speed_in_kmh** (*float*) – speed in kilometers/hour

Returns the speed in meters/second

Return type `float`

`taxi_simulator.helpers.random_position()`

Returns a random position inside the map.

Returns a point (longitude and latitude)

Return type `list`

5.1.5 taxi_simulator.passenger module

```
class taxi_simulator.passenger.PassengerAgent (agentjid, password, debug)
    Bases: spade.Agent.Agent

    add_strategy (strategy_class)
        Sets the strategy for the passenger agent.

        Parameters strategy_class (PassengerStrategyBehaviour) – The class to be
            used. Must inherit from PassengerStrategyBehaviour

    get_pickup_time ()
        Returns the time that the passenger was waiting to be picked up since it has been assigned to a taxi.

        Returns The time that the passenger was waiting to a taxi since it has been assigned.

        Return type float

    get_position ()
        Returns the current position of the passenger.

        Returns the coordinates of the current position of the passenger (lon, lat)

        Return type list

    get_value (key)
        Returns a stored value from the agent's knowledge base.

        Parameters key (str) – the name of the value

        Returns The object stored with the key

        Return type object

        Raises KeyError – if the key is not in the knowledge base

    get_waiting_time ()
        Returns the time that the agent was waiting for a taxi, from its creation until it gets into a taxi.

        Returns The time the passenger was waiting.

        Return type float

    has_value (key)
        Checks if a key is registered in the agent's knowledge base

        Parameters key (str) – the name of the value to be checked

        Returns whether the knowledge base has or not the key

        Return type bool

    is_in_destination ()
        Checks if the passenger has arrived to its destination.

        Returns whether the passenger is at its destination or not

        Return type bool

    request_path (origin, destination)
        Requests a path between two points (origin and destination) using the RouteAgent service.

        Parameters
```

- **origin** (*list*) – the coordinates of the origin of the requested path
- **destination** (*list*) – the coordinates of the end of the requested path

Returns A list of points that represent the path from origin to destination, the distance and the estimated duration

Return type list, float, float

set_id (*agent_id*)

Sets the agent identifier :param agent_id: The new Agent Id :type agent_id: str

set_position (*coords=None*)

Sets the position of the passenger. If no position is provided it is located in a random position.

Parameters **coords** (list) – a list coordinates (longitude and latitude)

set_target_position (*coords=None*)

Sets the target position of the passenger (i.e. its destination). If no position is provided the destination is setted to a random position.

Parameters **coords** (list) – a list coordinates (longitude and latitude)

store_value (*key, value*)

Stores a value (named by a key) in the agent's knowledge base that runs the behaviour. This allows the strategy to have persistent values between loops.

Parameters

- **key** (str) – the name of the value.
- **value** (object) – The object to be stored.

to_json ()

Serializes the main information of a passenger agent to a JSON format. It includes the id of the agent, its current position, the destination coordinates of the agent, the current status, the taxi that it has assigned (if any) and its waiting time.

Returns

a JSON doc with the main information of the passenger.

Example:

```
{
  "id": "cphillips",
  "position": [ 39.461327, -0.361839 ],
  "dest": [ 39.460599, -0.335041 ],
  "status": 24,
  "taxi": "ghiggins@127.0.0.1",
  "waiting": 13.45
}
```

Return type dict

total_time ()

Returns the time since the passenger was activated until it reached its destination.

Returns the total time of the passenger's simulation.

Return type float

class taxi_simulator.passenger.**PassengerStrategyBehaviour**

Bases: *taxi_simulator.utils.StrategyBehaviour*

Class from which to inherit to create a taxi strategy. You must overload the `_process()` method

Helper functions:

- `send_request()`
- `accept_taxi()`
- `refuse_taxi()`

accept_taxi (*taxi_aid*)

Sends an ACLMessage to a taxi to accept a travel proposal. It uses the REQUEST_PROTOCOL and the ACCEPT_PERFORMATIVE.

Parameters **taxi_aid** (spade.AID.aid) – The AgentID of the taxi

onStart ()

Initializes the logger and timers. Call to parent method if overloaded.

refuse_taxi (*taxi_aid*)

Sends an ACLMessage to a taxi to refuse a travel proposal. It uses the REQUEST_PROTOCOL and the REFUSE_PERFORMATIVE.

Parameters **taxi_aid** (spade.AID.aid) – The AgentID of the taxi

send_request (*content=None*)

Sends an ACLMessage to the coordinator to request a taxi. It uses the REQUEST_PROTOCOL and the REQUEST_PERFORMATIVE. If no content is set a default content with the passenger_id, origin and target coordinates is used.

Parameters **content** (dict) – Optional content dictionary

class taxi_simulator.passenger.TravelBehaviour

Bases: spade.Behaviour.Behaviour

This is the internal behaviour that manages the movement of the passenger. It is triggered when the taxi informs the passenger that it is going to the passenger's position until the passenger is dropped in its destination.

onStart ()

5.1.6 taxi_simulator.protocol module

protocol and performative constants

5.1.7 taxi_simulator.route module

class taxi_simulator.route.RouteAgent (*agentjid, password, debug*)

Bases: spade.Agent.Agent

The RouteAgent receives request for paths, queries an OSRM server and returns the information. It also caches the queries to avoid overloading the OSRM server.

class RequestRouteBehaviour

Bases: spade.Behaviour.Behaviour

This cyclic behaviour listens for route requests from other agents. When a message is received it answers with the path.

onEnd ()

onStart ()

get_route (*origin, destination*)

Checks the cache for a path, if not found then it queries the OSRM server.

Parameters

- **origin** (*list*) – origin coordinate (longitude, latitude)
- **destination** (*list*) – target coordinate (longitude, latitude)

Returns a dict with three keys: path, distance and duration

Return type dict

load_cache ()

Loads the cache from file.

persist_cache ()

Persists the cache to a JSON file.

static request_route_to_server (*origin, destination*)

Queries the OSRM for a path.

Parameters

- **origin** (*list*) – origin coordinate (longitude, latitude)
- **destination** (*list*) – target coordinate (longitude, latitude)

Returns list, float, float = the path, the distance of the path and the estimated duration

5.1.8 taxi_simulator.scenario module

class taxi_simulator.scenario.**Scenario** (*filename, debug_level=None*)

Bases: object

A scenario object reads a file with a JSON representation of a scenario and is used to create the participant agents.

static create_agent (*name, password, position, target, debug_level*)

Create an agent of type `cls`.

Parameters

- **cls** (*class*) – class of the agent
- **name** (*str*) – name of the agent
- **password** (*str*) – password of the agent
- **position** (*list*) – initial coordinates of the agent
- **target** (*list optional*) – destination coordinates of the agent
- **debug_level** – level of debug (None or ‘always’)

Returns the created agent

Return type object

classmethod create_agents_batch (*type_, number, coordinator*)

Creates a batch of agents.

Parameters

- **cls** (*class*) – class of the agents
- **type** (*str*) – whether the agent is a “taxi” or a “passenger”
- **number** (*int*) – size of the batch
- **((coordinator)** – obj: ‘CoordinatorAgent’): the coordinator agent

5.1.9 taxi_simulator.simulator module

class taxi_simulator.simulator.**FlaskBackend** (*command_queue*, *host*='0.0.0.0', *port*=5000, *debug*=False)

Bases: object

generate (*args, **kwargs)

class taxi_simulator.simulator.**SimulationConfig**

Bases: object

Dataclass to store the *Simulator* config

class taxi_simulator.simulator.**Simulator** (*config*)

Bases: object

The Simulator. It manages all the simulation processes. Tasks done by the simulator at initialization:

1. Create the XMPP server
2. Run the SPADE backend
3. Run the coordinator and route agents.
4. Create agents passed as parameters (if any).
5. Create agents defined in scenario (if any).
6. Create a backend web server (*FlaskBackend*) to listen for async commands.

After these tasks are done in the Simulator constructor, the simulation is started when the *run()* method is called.

collect_stats()

Collects stats from all participant agents and from the simulation and stores it in three dataframes.

get_stats()

Returns the dataframes collected by *collect_stats()*

Returns average df, passengers df and taxi df

Return type pandas.DataFrame, pandas.DataFrame, pandas.DataFrame

is_simulation_finished()

Checks if the simulation is finished. A simulation is finished if the max simulation time has been reached or when the coordinator says it.

Returns whether the simulation is finished or not.

Return type bool

print_stats()

Prints the dataframes collected by *collect_stats()*.

process_queue()

Queries the command queue if there is any command to execute. If true, runs the command.

At the moment the only command available is to create new taxis and passengers.

run()

Starts the simulation (tells the coordinator agent to start the simulation).

stop()

Finishes the simulation and prints simulation stats. Tasks done when a simulation is stopped:

1. Terminate web backend.

2. Stop participant agents.
3. Print stats.
4. Stop Route agent.
5. Stop Coordinator agent.
6. Shutdown SPADE backend.
7. Shutdown XMPP server.

time_is_out ()

Checks if the max simulation time has been reached.

Returns whether the max simulation time has been reached or not.

Return type bool

write_excel (filename)

Writes the collected data by `collect_stats()` in an excel file.

Parameters **filename** (str) – name of the excel file.

write_file (filename, fileformat='json')

Writes the dataframes collected by `collect_stats()` in JSON or Excel format.

Parameters

- **filename** (str) – name of the output file to be written.
- **fileformat** (str) – format of the output file. Choices: json or excel

write_json (filename)

Writes the collected data by `collect_stats()` in a json file.

Parameters **filename** (str) – name of the json file.

5.1.10 taxi_simulator.strategies module

class taxi_simulator.strategies.**AcceptAlwaysStrategyBehaviour**

Bases: `taxi_simulator.taxi.TaxiStrategyBehaviour`

The default strategy for the Taxi agent. By default it accepts every request it receives if available.

class taxi_simulator.strategies.**AcceptFirstRequestTaxiBehaviour**

Bases: `taxi_simulator.passenger.PassengerStrategyBehaviour`

The default strategy for the Passenger agent. By default it accepts the first proposal it receives.

class taxi_simulator.strategies.**DelegateRequestTaxiBehaviour**

Bases: `taxi_simulator.coordinator.CoordinatorStrategyBehaviour`

The default strategy for the Coordinator agent. By default it delegates all requests to all taxis.

5.1.11 taxi_simulator.taxi module

class taxi_simulator.taxi.**TaxiAgent** (agentjid, password, debug)

Bases: `spade.Agent.Agent`

class **MovingBehaviour** (period, timestart=None)

Bases: `spade.Behaviour.PeriodicBehaviour`

This is the internal behaviour that manages the movement of the taxi. It is triggered when the taxi has a new destination and the periodic tick is recomputed at every step to show a fine animation. This moving behaviour includes to update the taxi coordinates as it moves along the path at the specified speed.

add_strategy (*strategy_class*)

Sets the strategy for the taxi agent.

Parameters **strategy_class** (*TaxiStrategyBehaviour*) – The class to be used.
Must inherit from *TaxiStrategyBehaviour*

arrived_to_destination ()

MVC view executed when the taxi has arrived to its destination. It recomputes the new destination and path if picking up a passenger or drops it and goes to WAITING status again.

Returns an empty template and data. This view is a JSON request, so it does not render any new template.

Return type None, dict

cancel_passenger (*data=None*)

Sends a message to the current assigned passenger to cancel the assignment.

Parameters **data** (*dict optional*) – Complementary info about the cancellation

drop_passenger ()

Drops the passenger that the taxi is carrying in the current location.

get_position ()

Returns the current position of the passenger.

Returns the coordinates of the current position of the passenger (lon, lat)

Return type list

get_value (*key*)

Returns a stored value from the agent's knowledge base.

Parameters **key** (*str*) – the name of the value

Returns The object stored with the key

Return type object

Raises *KeyError* – if the key is not in the knowledge base

has_value (*key*)

Checks if a key is registered in the agent's knowledge base

Parameters **key** (*str*) – the name of the value to be checked

Returns whether the knowledge base has or not the key

Return type bool

inform_passenger (*status, data=None*)

Sends a message to the current assigned passenger to inform her about a new status.

Parameters

- **status** (*int*) – The new status code
- **data** (*dict optional*) – complementary info about the status

is_in_destination ()

Checks if the taxi has arrived to its destination.

Returns whether the taxi is at its destination or not

Return type bool

move_to (*dest*)

Moves the taxi to a new destination.

Parameters **dest** (*list*) – the coordinates of the new destination (in lon, lat format)

Raises `AlreadyInDestination` – if the taxi is already in the destination coordinates.

request_path (*origin, destination*)

Requests a path between two points (origin and destination) using the RouteAgent service.

Parameters

- **origin** (*list*) – the coordinates of the origin of the requested path
- **destination** (*list*) – the coordinates of the end of the requested path

Returns A list of points that represent the path from origin to destination, the distance and the estimated duration

Return type list, float, float

Examples

```
>>> path, distance, duration = self.request_path(origin=[0,0], destination=[1,
↪1])
>>> print(path)
[[0,0], [0,1], [1,1]]
>>> print(distance)
2.0
>>> print(duration)
3.24
```

set_id (*agent_id*)

Sets the agent identifier

Parameters **agent_id** (*str*) – The new Agent Id

set_position (*coords=None*)

Sets the position of the taxi. If no position is provided it is located in a random position.

Parameters **coords** (*list*) – a list coordinates (longitude and latitude)

set_speed (*speed_in_kmh*)

Sets the speed of the taxi.

Parameters **speed_in_kmh** (*float*) – the speed of the taxi in km per hour

step ()

Advances one step in the simulation

store_value (*key, value*)

Stores a value (named by a key) in the agent's knowledge base that runs the behaviour. This allows the strategy to have persistent values between loops.

Parameters

- **key** (*str*) – the name of the value.
- **value** (*object*) – The object to be stored.

to_json()

Serializes the main information of a taxi agent to a JSON format. It includes the id of the agent, its current position, the destination coordinates of the agent, the current status, the speed of the taxi (in km/h), the path it is following (if any), the passenger that it has assigned (if any), the number of assignments if has done and the distance that the taxi has traveled.

Returns

a JSON doc with the main information of the taxi.

Example:

```
{
  "id": "cphillips",
  "position": [ 39.461327, -0.361839 ],
  "dest": [ 39.460599, -0.335041 ],
  "status": 24,
  "speed": 1000,
  "path": [[0,0], [0,1], [1,0], [1,1], ...],
  "passenger": "ghiggins@127.0.0.1",
  "assignments": 2,
  "distance": 3481.34
}
```

Return type dict

class taxi_simulator.taxi.**TaxiStrategyBehaviour**

Bases: *taxi_simulator.utils.StrategyBehaviour*

Class from which to inherit to create a taxi strategy. You must overload the `_process()` method

Helper functions:

- *pick_up_passenger()*
- *send_proposal()*
- *cancel_proposal()*

cancel_proposal (*passenger_id*, *content=None*)

Send an ACLMessage to cancel a proposal. If the content is empty the proposal is sent without content.

Parameters

- **passenger_id** (*str*) – the id of the passenger
- **content** (*dict optional*) – the optional content of the message

onStart ()

pick_up_passenger (*passenger_id*, *origin*, *dest*)

Starts a TRAVEL_PROTOCOL to pick up a passenger and get him to his destination. It automatically launches all the travelling process until the passenger is delivered. This travelling process includes to update the taxi coordinates as it moves along the path at the specified speed.

Parameters

- **passenger_id** (*str*) – the id of the passenger
- **origin** (*list*) – the coordinates of the current location of the passenger
- **dest** (*list*) – the coordinates of the target destination of the passenger

send_proposal (*passenger_id*, *content=None*)

Send an `ACLMessage` with a proposal to a passenger to pick up him. If the content is empty the proposal is sent without content.

Parameters

- **passenger_id** (*str*) – the id of the passenger
- **content** (*dict optional*) – the optional content of the message

5.1.12 taxi_simulator.utils module

class `taxi_simulator.utils.RequestRouteBehaviour` (*msg*, *origin*, *destination*)

Bases: `spade.Behaviour.OneShotBehaviour`

A one-shot behaviour that is executed to request for a new route to the route agent.

class `taxi_simulator.utils.StrategyBehaviour`

Bases: `spade.Behaviour.Behaviour`

The behaviour that all parent strategies must inherit from. It complies with the Strategy Pattern.

get_value (*key*)

Returns a stored value from the agent's knowledge base.

Parameters **key** (*str*) – the name of the value

Returns the object stored with key

Return type object

Raises `KeyError` – if the key is not in the knowledge base

has_value (*key*)

Checks if a key is registered in the agent's knowledge base

Parameters **key** (*str*) – the name of the value to be checked

Returns if the key is in the agent's knowledge base

Return type bool

receive (*timeout=5*)

Waits for a message until timeout is done. If a message is received the method returns immediately. If the time has passed and no message has been received, it returns None.

Parameters **timeout** (*int optional*) – number of seconds to wait for a message

Returns a message or None

Return type `spade.ACLMessage.ACLMessage` or None

send (*message*)

Sends an `spade.ACLMessage.ACLMessage`

Parameters **message** (`spade.ACLMessage.ACLMessage`) – the message to be sent

store_value (*key*, *value*)

Stores a value (named by a key) in the agent's knowledge base that runs the behaviour. This allows the strategy to have persistent values between loops.

Parameters

- **key** (*str*) – the name of the value
- **value** (*object*) – The object to be stored

`timeout_receive (timeout=5)`

`taxi_simulator.utils.chunk_path (path, speed_in_kmh)`
Splits the path into smaller chunks taking into account the speed.

Parameters

- **path** (*list*) – the original path. A list of points (lon, lat)
- **speed_in_kmh** (*float*) – the speed in km per hour at which the path is being traveled.

Returns a new path equivalent (to the first one), that has at least the same number of points.

Return type list

`taxi_simulator.utils.crossdomain (origin=None, methods=None, headers=None, max_age=21600, attach_to_all=True, automatic_options=True)`
Decorator to allow the cross-domain situation in web requests.

`taxi_simulator.utils.load_class (class_path)`
Tricky method that imports a class from a string.

Parameters **class_path** (*str*) – the path where the class to be imported is.

Returns the class imported and ready to be instantiated.

Return type class

`taxi_simulator.utils.request_path (agent, origin, destination)`
Sends a message to the RouteAgent to request a path

Parameters

- **agent** – the agent who is requesting the path
- **origin** (*list*) – a list with the origin coordinates [longitude, latitude]
- **destination** (*list*) – a list with the target coordinates [longitude, latitude]

Returns a list of points (longitude and latitude) representing the path, the distance of the path in meters, a estimation of the duration of the path

Return type list, float, float

Examples

```
>>> path, distance, duration = request_path(an_agent, origin=[0,0],
↪destination=[1,1])
>>> print (path)
[[0,0], [0,1], [1,1]]
>>> print (distance)
2.0
>>> print (duration)
3.24
```

`taxi_simulator.utils.status_to_str (status_code)`
Translates an int status code to a string that represents the status

Parameters **status_code** (*int*) – the code of the status

Returns the string that represents the status

Return type str

`taxi_simulator.utils.timeout_receive(agent, timeout)`
makes an agent to wait for a message until a timeout

Parameters

- **agent** – the agent who is receiving the message
- **timeout** (*int*) – the number of seconds to wait for the message

Returns a received message or None

Return type `spade.ACLMessage.ACLMessage`

`taxi_simulator.utils.unused_port(hostname)`
Return a port that is unused on the current host.

5.1.13 Module contents

Top-level package for Taxi Simulator.

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given. You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at https://github.com/javipalanca/taxi_simulator/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” and “help wanted” is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with “enhancement” and “help wanted” is open to whoever wants to implement it.

6.1.4 Write Documentation

Taxi Simulator could always use more documentation, whether as part of the official Taxi Simulator docs, in docstrings, or even on the web in blog posts, articles, and such.

6.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/javipalanca/taxi_simulator/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up *taxi_simulator* for local development.

1. Fork the *taxi_simulator* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/taxi_simulator.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv taxi_simulator
$ cd taxi_simulator/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 taxi_simulator tests
$ python setup.py test or py.test
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, 3.4 and 3.5, and for PyPy. Check https://travis-ci.org/javipalanca/taxi_simulator/pull_requests and make sure that the tests pass for all supported Python versions.

6.4 Tips

To run a subset of tests:

```
$ py.test tests.test_taxi_simulator
```


7.1 Development Lead

- Javi Palanca <jpalanca@gmail.com>

7.2 Contributors

None yet. Why not be the first?

8.1 0.3 ()

- Documentation highly improved.
- Helper functions added and refined.
- Javascript framework included: VueJS
- Routes centralized with a Route agent.
- UI improved.

8.2 0.2 (2017-11-15)

- Added scenario loading feature.

8.3 0.1.3 (2017-11-15)

- Fixed minor bugs.

8.4 0.1.1 (2017-11-14)

- Added documentation.

8.5 0.1.0 (2017-11-03)

- First release on PyPI.

CHAPTER 9

Indices and tables

- `genindex`
- `modindex`
- `search`

Bibliography

[GangOfFour95] 5. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns, Elements of Reusable Object Oriented Software. Addison-Wesley, 1995.

t

- `taxi_simulator`, [56](#)
- `taxi_simulator.cli`, [39](#)
- `taxi_simulator.coordinator`, [39](#)
- `taxi_simulator.helpers`, [43](#)
- `taxi_simulator.passenger`, [45](#)
- `taxi_simulator.protocol`, [47](#)
- `taxi_simulator.route`, [47](#)
- `taxi_simulator.scenario`, [48](#)
- `taxi_simulator.simulator`, [49](#)
- `taxi_simulator.strategies`, [50](#)
- `taxi_simulator.taxi`, [50](#)
- `taxi_simulator.utils`, [54](#)

A

[accept_taxi\(\)](#) (taxi_simulator.passenger.PassengerStrategyBehaviour taxi_simulator.coordinator), 43
[method](#), 47
[AcceptAlwaysStrategyBehaviour](#) (class in taxi_simulator.strategies), 50
[AcceptFirstRequestTaxiBehaviour](#) (class in taxi_simulator.strategies), 50
[add_passenger\(\)](#) (taxi_simulator.coordinator.CoordinatorAgent method), 39
[add_strategy\(\)](#) (taxi_simulator.coordinator.CoordinatorAgent method), 39
[add_strategy\(\)](#) (taxi_simulator.passenger.PassengerAgent method), 45
[add_strategy\(\)](#) (taxi_simulator.taxi.TaxiAgent method), 51
[add_taxi\(\)](#) (taxi_simulator.coordinator.CoordinatorAgent method), 39
[AlreadyInDestination](#), 43
[are_close\(\)](#) (in module taxi_simulator.helpers), 44
[arrived_to_destination\(\)](#) (taxi_simulator.taxi.TaxiAgent method), 51

B

[build_aid\(\)](#) (in module taxi_simulator.helpers), 44

C

[cancel_passenger\(\)](#) (taxi_simulator.taxi.TaxiAgent method), 51
[cancel_proposal\(\)](#) (taxi_simulator.taxi.TaxiStrategyBehaviour method), 53
[chunk_path\(\)](#) (in module taxi_simulator.utils), 55
[clean_controller\(\)](#) (taxi_simulator.coordinator.CoordinatorAgent method), 40
[collect_stats\(\)](#) (taxi_simulator.simulator.Simulator method), 49
[content_to_json\(\)](#) (in module taxi_simulator.helpers), 44
[CoordinatorAgent](#) (class in taxi_simulator.coordinator), 39

[CoordinatorStrategyBehaviour](#) (class in taxi_simulator.strategies), 50
[create_agent\(\)](#) (taxi_simulator.scenario.Scenario static method), 48
[create_agents_batch\(\)](#) (taxi_simulator.scenario.Scenario class method), 48
[crossdomain\(\)](#) (in module taxi_simulator.utils), 55

D

[DelegateRequestTaxiBehaviour](#) (class in taxi_simulator.strategies), 50
[distance_in_meters\(\)](#) (in module taxi_simulator.helpers), 44
[drop_passenger\(\)](#) (taxi_simulator.taxi.TaxiAgent method), 51

E

[entities_controller\(\)](#) (taxi_simulator.coordinator.CoordinatorAgent method), 40

F

[FlaskBackend](#) (class in taxi_simulator.simulator), 49

G

[generate\(\)](#) (taxi_simulator.simulator.FlaskBackend method), 49
[generate_tree\(\)](#) (taxi_simulator.coordinator.CoordinatorAgent method), 41
[get_passenger_agents\(\)](#) (taxi_simulator.coordinator.CoordinatorStrategyBehaviour method), 43
[get_passenger_stats\(\)](#) (taxi_simulator.coordinator.CoordinatorAgent method), 41
[get_pickup_time\(\)](#) (taxi_simulator.passenger.PassengerAgent method), 45
[get_position\(\)](#) (taxi_simulator.passenger.PassengerAgent method), 45
[get_position\(\)](#) (taxi_simulator.taxi.TaxiAgent method), 51
[get_route\(\)](#) (taxi_simulator.route.RouteAgent method), 47

`get_simulation_time()` (taxi_simulator.coordinator.CoordinatorAgent method), 41

`get_stats()` (taxi_simulator.coordinator.CoordinatorAgent method), 41

`get_stats()` (taxi_simulator.simulator.Simulator method), 49

`get_taxi_agents()` (taxi_simulator.coordinator.CoordinatorStrategyBehaviour method), 43

`get_taxi_stats()` (taxi_simulator.coordinator.CoordinatorAgent method), 41

`get_value()` (taxi_simulator.coordinator.CoordinatorAgent method), 41

`get_value()` (taxi_simulator.passenger.PassengerAgent method), 45

`get_value()` (taxi_simulator.taxi.TaxiAgent method), 51

`get_value()` (taxi_simulator.utils.StrategyBehaviour method), 54

`get_waiting_time()` (taxi_simulator.passenger.PassengerAgent method), 45

H

`has_value()` (taxi_simulator.coordinator.CoordinatorAgent method), 42

`has_value()` (taxi_simulator.passenger.PassengerAgent method), 45

`has_value()` (taxi_simulator.taxi.TaxiAgent method), 51

`has_value()` (taxi_simulator.utils.StrategyBehaviour method), 54

I

`index_controller()` (taxi_simulator.coordinator.CoordinatorAgent method), 42

`inform_passenger()` (taxi_simulator.taxi.TaxiAgent method), 51

`is_in_destination()` (taxi_simulator.passenger.PassengerAgent method), 45

`is_in_destination()` (taxi_simulator.taxi.TaxiAgent method), 51

`is_simulation_finished()` (taxi_simulator.coordinator.CoordinatorAgent method), 42

`is_simulation_finished()` (taxi_simulator.simulator.Simulator method), 49

K

`kmh_to_ms()` (in module taxi_simulator.helpers), 44

L

`load_cache()` (taxi_simulator.route.RouteAgent method), 48

`load_class()` (in module taxi_simulator.utils), 55

M

`move_to()` (taxi_simulator.taxi.TaxiAgent method), 52

O

`onEnd()` (taxi_simulator.route.RouteAgent.RequestRouteBehaviour method), 47

`onStart()` (taxi_simulator.coordinator.CoordinatorStrategyBehaviour method), 43

`onStart()` (taxi_simulator.passenger.PassengerStrategyBehaviour method), 47

`onStart()` (taxi_simulator.passenger.TravelBehaviour method), 47

`onStart()` (taxi_simulator.route.RouteAgent.RequestRouteBehaviour method), 47

`onStart()` (taxi_simulator.taxi.TaxiStrategyBehaviour method), 53

P

`passenger_agents` (taxi_simulator.coordinator.CoordinatorAgent attribute), 42

`PassengerAgent` (class in taxi_simulator.passenger), 45

`PassengerStrategyBehaviour` (class in taxi_simulator.passenger), 46

`PathRequestException`, 43

`persist_cache()` (taxi_simulator.route.RouteAgent method), 48

`pick_up_passenger()` (taxi_simulator.taxi.TaxiStrategyBehaviour method), 53

`print_stats()` (taxi_simulator.simulator.Simulator method), 49

`process_queue()` (taxi_simulator.simulator.Simulator method), 49

R

`random_position()` (in module taxi_simulator.helpers), 44

`receive()` (taxi_simulator.utils.StrategyBehaviour method), 54

`refuse_taxi()` (taxi_simulator.passenger.PassengerStrategyBehaviour method), 47

`request_path()` (in module taxi_simulator.utils), 55

`request_path()` (taxi_simulator.coordinator.CoordinatorAgent method), 42

`request_path()` (taxi_simulator.passenger.PassengerAgent method), 45

`request_path()` (taxi_simulator.taxi.TaxiAgent method), 52

`request_route_to_server()` (taxi_simulator.route.RouteAgent static method), 48

`RequestRouteBehaviour` (class in taxi_simulator.utils), 54

`RouteAgent` (class in taxi_simulator.route), 47

`RouteAgent.RequestRouteBehaviour` (class in taxi_simulator.route), 47

`run()` (taxi_simulator.simulator.Simulator method), 49

`run_controller()` (taxi_simulator.coordinator.CoordinatorAgent method), 42

run_simulation() (taxi_simulator.coordinator.CoordinatorAgent method), 42

S

Scenario (class in taxi_simulator.scenario), 48

send() (taxi_simulator.utils.StrategyBehaviour method), 54

send_proposal() (taxi_simulator.taxi.TaxiStrategyBehaviour method), 53

send_request() (taxi_simulator.passenger.PassengerStrategyBehaviour method), 47

set_id() (taxi_simulator.passenger.PassengerAgent method), 46

set_id() (taxi_simulator.taxi.TaxiAgent method), 52

set_position() (taxi_simulator.passenger.PassengerAgent method), 46

set_position() (taxi_simulator.taxi.TaxiAgent method), 52

set_speed() (taxi_simulator.taxi.TaxiAgent method), 52

set_strategies() (taxi_simulator.coordinator.CoordinatorAgent method), 42

set_target_position() (taxi_simulator.passenger.PassengerAgent method), 46

SimulationConfig (class in taxi_simulator.simulator), 49

Simulator (class in taxi_simulator.simulator), 49

status_to_str() (in module taxi_simulator.utils), 55

step() (taxi_simulator.taxi.TaxiAgent method), 52

stop() (taxi_simulator.simulator.Simulator method), 49

stop_agents() (taxi_simulator.coordinator.CoordinatorAgent method), 43

store_value() (taxi_simulator.coordinator.CoordinatorAgent method), 43

store_value() (taxi_simulator.passenger.PassengerAgent method), 46

store_value() (taxi_simulator.taxi.TaxiAgent method), 52

store_value() (taxi_simulator.utils.StrategyBehaviour method), 54

StrategyBehaviour (class in taxi_simulator.utils), 54

T

taxi_agents (taxi_simulator.coordinator.CoordinatorAgent attribute), 43

taxi_simulator (module), 56

taxi_simulator.cli (module), 39

taxi_simulator.coordinator (module), 39

taxi_simulator.helpers (module), 43

taxi_simulator.passenger (module), 45

taxi_simulator.protocol (module), 47

taxi_simulator.route (module), 47

taxi_simulator.scenario (module), 48

taxi_simulator.simulator (module), 49

taxi_simulator.strategies (module), 50

taxi_simulator.taxi (module), 50

taxi_simulator.utils (module), 54

TaxiAgent (class in taxi_simulator.taxi), 50

TaxiAgent.MovingBehaviour (class in taxi_simulator.taxi), 50

TaxiStrategyBehaviour (class in taxi_simulator.taxi), 53

time_is_out() (taxi_simulator.simulator.Simulator method), 50

timeout_receive() (in module taxi_simulator.utils), 55

timeout_receive() (taxi_simulator.utils.StrategyBehaviour method), 55

to_json() (taxi_simulator.passenger.PassengerAgent method), 46

to_json() (taxi_simulator.taxi.TaxiAgent method), 52

total_time() (taxi_simulator.passenger.PassengerAgent method), 46

TravelBehaviour (class in taxi_simulator.passenger), 47

U

unused_port() (in module taxi_simulator.utils), 56

W

write_excel() (taxi_simulator.simulator.Simulator method), 50

write_file() (taxi_simulator.simulator.Simulator method), 50

write_json() (taxi_simulator.simulator.Simulator method), 50