Wiki »

1. Installation Instructions For AgentDrive

Running from source code:

The following steps show how to run AgentDrive from source code in Netbeans. However, any other Integrated Development Environment can be used instead.

1. Obtain the source code and extract the zip to any location you want.

2. Start Eclipse and create a new Maven Project (File ☐ New Project ☐ Maven ☐ Project with existing POM) and click 'Next'. Then, Click 'Finish'.

3. Browse to the extracted location of the project and select the 'highway' project. Click 'Open Project'.

4. Repeat the steps 2 and 3 and select 'Simulator-Lite' and finish creating the second project.

5. Then again repeat the steps 2 and 3 and select 'Simulator' and finish creating the third project.

6. Create .m2 Folder in "C:\Users\XYZ" (For Windows Users) or "Users->XYZ" (For Linux or MacOs users) and copy the "settings.xml" from the downloaded zip file.

7. Go to Netbeans and select "AgentDrive" in Projects tab. Right click and select "Resolve Project Problems". This will download dependencies for the project and try to resolve the problems.

8. Repeat the step 7 for the project "Simulator-Lite" and "Simulator". This will make your projects error-free.

9. Specify run parameters(Run-> Set project Configuration-> Customise-> Run) for the three projects as mentioned below:

   a. **For AgentDrive**

      i. MainClass: cz.agents.alite.Main

      ii. Program Arguments:

         1. cz.agents.highway.creator.DashBoardController
         2. settings/groovy/local/martin.groovy

> > *The first argument is Creator to be used and second argument is configuration file.*
> > *If the second argument is not used the default config file is used i.e. settings/groovy/highway.groovy.*
>
> *It is recommended not to change the default config file if not necessary, you can create a copy of it in subfolder 'local' and use it instead.*

   b. **For Simulator-lite**

      i. MainClass: cz.agents.agentdrive.simulator.lite.creator.SimulatorCreator

      ii. Program Arguments:

         1. cz.agents.agentdrive.simulator.lite.creator.SimulatorCreator

      iii. Working Directory: $MODULE_DIR$

> > *The first argument is Creator to be used.*
>
> *Working directory should be set to the module directory if you have imported simulator-lite as a module of the AgentDrive (highway) project.*

   c. **For Simulator (OpenDS)**

      i. MainClass: eu.opends.main.Simulator

      ii. Program Arguments:

         1. assets/DrivingTasks/Projects/Highway/osm.xml

      iii. Working Directory: $MODULE_DIR$

> > *Note that you should run AgentDrive(Highway) first, then run the simulator.*
>
> *If you do not use the .sh script to start the simulator you can choose Empty in AgentDrive configuration and run the simulator manually.*

10. Click 'Build and Run' option in Run tab in Netbeans for the project 'AgentDrive'. Then run the project. After it has run successfully, a Java window appears showing "Initialising" in the middle. This means it is waiting for the inputs from either "Simulator-Lite" or "Simulator".

11. Try to repeat the step 9 for "Simulator-Lite". Now you can see the cars moving and planning autonomous maneuvers in both AgentDrive and Simulator-Lite java windows.

2. Description

a. Usage of Classes

1. ***Agent Class:***

   An agent class defined under the package "cz.agents.highway.agent" is responsible for defining plans for an agent. And we define plans for an agent in terms of Waypoints. We can re-define the way of implementation to meet our expectations from an agent.
   An agent class need to extend class "Agent" under same package. This enables the agent to sense Position, Velocity and other parameters of their neighbouring vehicles. Moreover, the agents are initialised for each vehicle specified with unique id in a XML file. For example; for the scenario straight-highway the file is located in "SourcePackages/Other Sources/src/main/resources/nets.highway-straight/highway-straight.rou.xml ". And as we can see in the directory that there is also a file named "highway-straight.net". This basically initialises a lane with several parameters like id, index, speed, length, shape (specified in the form of a rectangle where the first two parameters are the bottom left co-ordinates and the other two are top-right co-ordinates.

2. ***Creator Package:***

   This package defines three classes named, DashBoardController, DefaultCreator and Main Class. DashBoardController extends DefaultCreator Class; which is responsible for creating the simulation environment as defined in the file specified in the second run argument of AgentDrive. The configuration file referred in run argument say (test.groovy) of AgentDrive is defined at location "settings/groovy/local/test.groovy". This file saves parameters for simulating the environment and stores several constants like acceleration, deceleration, simulation timestep, simulation speed and so on.

   a. *DashBoardController*: It manages launching of simulators, and their synchronization with AgentDrive and sending simulator appropriate plans and updates. For this we need to define methods to initialise traffic and even distribution of vehicles, create agent for every single vehicle.

   b. *DefaultCreator*: It defines methods to run then simulation, create visualisation and register various layers of visualisations (Fps layer, Help Layer), create environment and so on.

3. ***Package Highway environment road net:***

   This package consists of classes that define connection, edges, junction etc. between the lanes on a road network. For this. hash maps are required to map a string to edge or junction. Also , kd-Trees are created and filled with point - lane pairs for fast look up of the cars current lane based on its x,y coordinates only.

4. ***Package Highway Maneuver:***

   Several classes defining various possible types of maneuvers like straight maneuver, acceleration maneuver, deacceleration maneuver and so on are defined in this package. The maneuvers provide waypoints to the vehicle and the vehicle then reacts to the waypoints given by following them.
   So several parameters like acceleration , duration of maneuever, entry lane, exit lane and so on are required when initialising an instance of any of the possible maneuvers. All the maneuvers extend the class CarManeuver and therefore it would be good to instantiate an object of the required maneuvers and then refer it to by initialising an instance of CarManeuver class. For example:

   ```
   CarManeuver sm= new Straighmaneuver();
   ```

   As required for the agent, we can later change the maneuvering types for an agent.

5. ***Package highway.storage.plan:***

   This package acts as a storage for the Actions, ManeuverAction, Plans, WayPointActions etc. Thus storage classes facilitate retrieving the CarId, TimeStamp, Plans, Speed, Position and so on.

b. Main Methods used

Important Classes:

• *Agent Class*: This class is the super class for any type of agent that plans and provide WayPoints for the vehicles.

• *DashBoardController*: It is responsible for distribution of vehicles and defining initial conditions for a vehicle.

• *ManeuverTranslator*: This class translates maneuvers generated by sub-classes of Agent to waypoints

• *RouteNavigator*: This Class used for car navigation on given route.

• *CarManeuver*: It acts as a super class for several possible types of maneuvering.

• *State*: It stores the carId of the vehicles in the neighbourhood of the current vehicles.

• *VehicleSensor*: It provides methods to sense the current car or a collection of cars and further return the position, velocity and other aspects related to the vehicles.

• *Connection, Junction, Edge, lane*: It defines a structure holding connections through junction loaded from sumo.net.xml file.

• *Edge, Junction, Lane, Request*: It is basically a structure holding data about an (edge/junction/lane/Request data of a junction) loaded from sumo.net.xml file.

• *Network*: It provides the following data: edges, junctions, lanes, connections, tunnels, bridges. It also provides a converter from x,y coordinates to a specific lane.

Important methods:

• *Agent React*: The method defined in class extended by AgentClass. It is responsible for returning Waypoints using the class WPAction (which extend
Action Class).

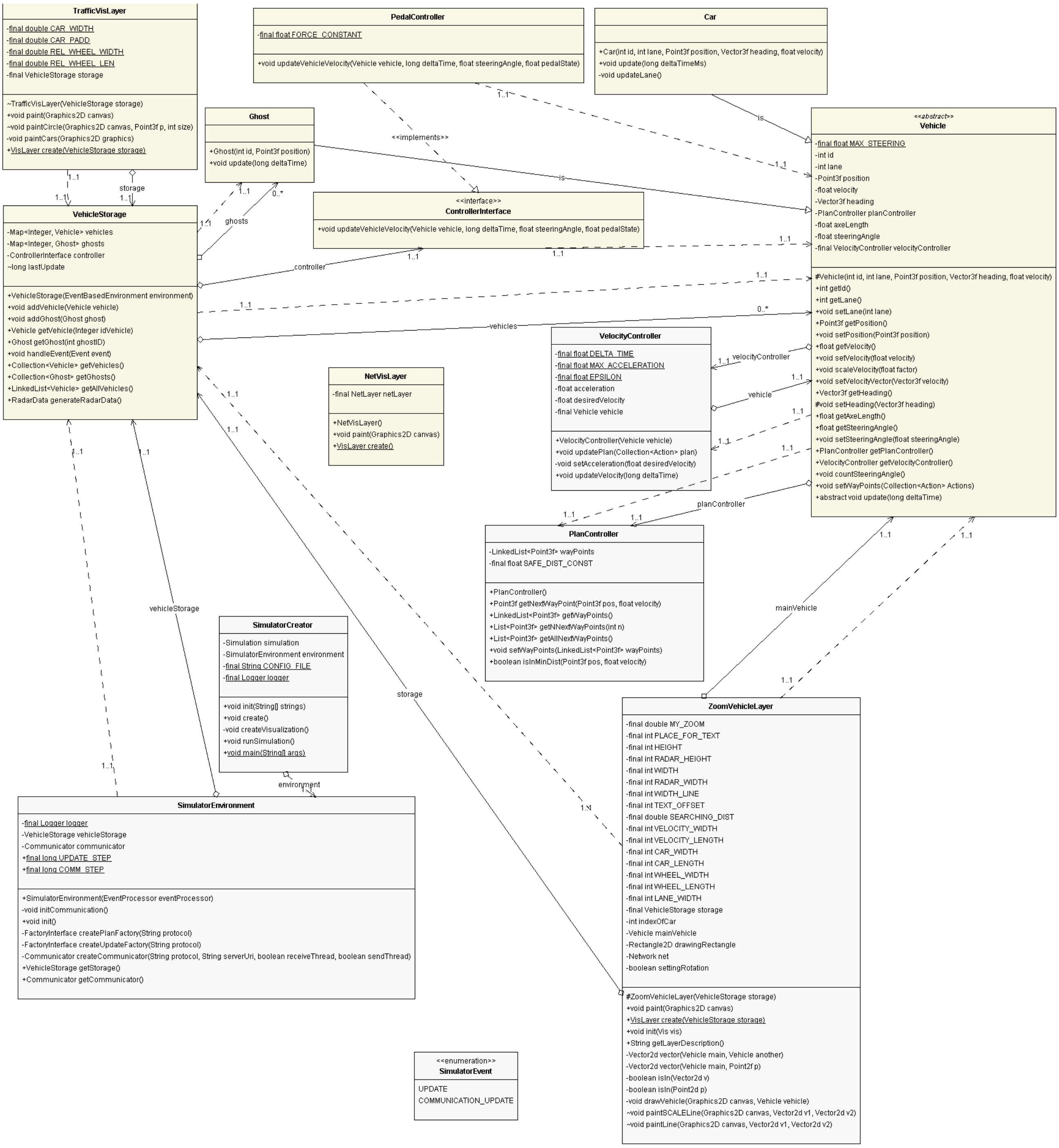
• Translate: Translates a maneuver to waypoints.

• *Plan_maneouvre*: This method tries to pre-plan and predict possible co-ordinates on each state change.

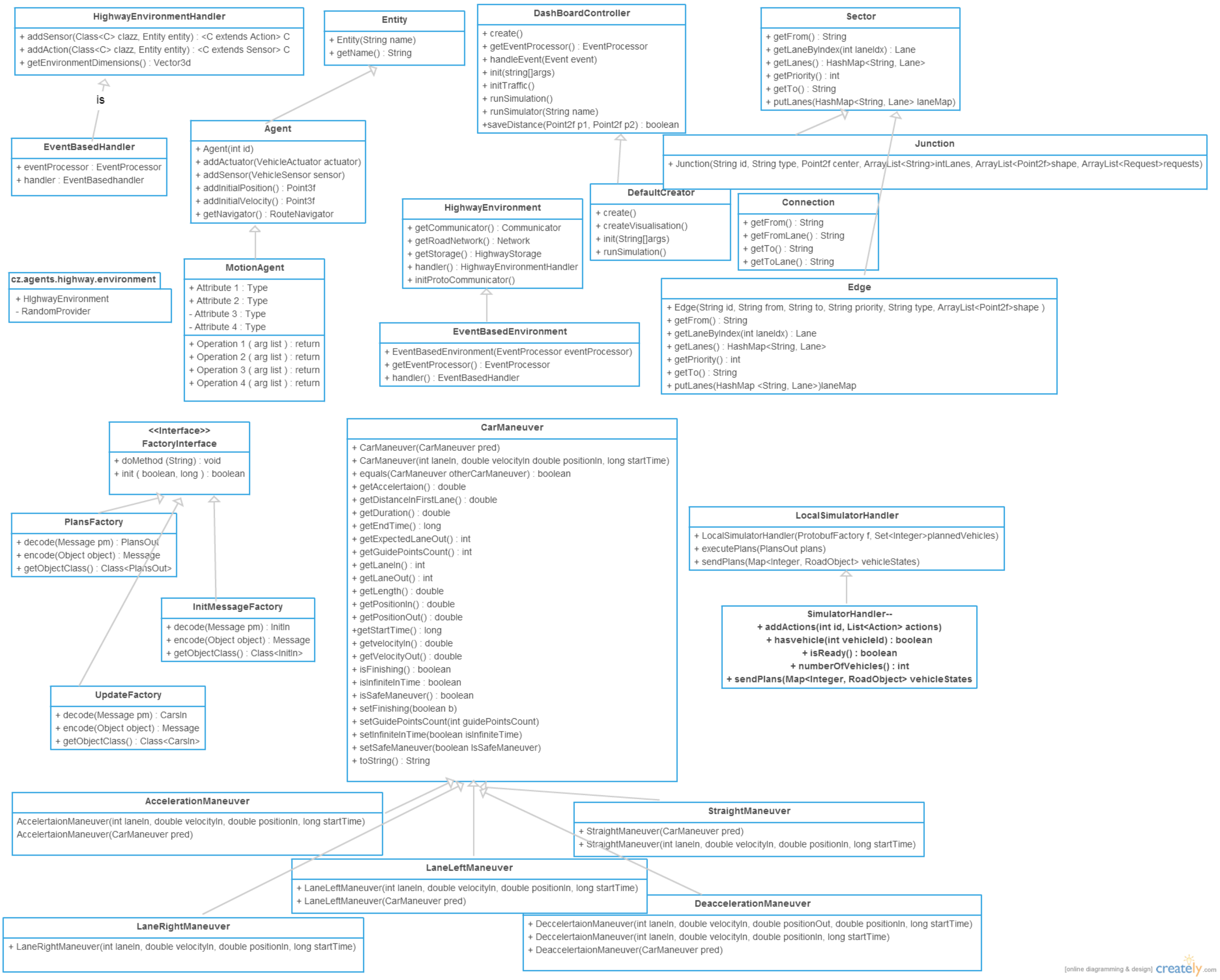
• *addSensor*: Adds a sensor to the motionAgent and further provides the possibility for actuators to act , given a list of actions.

c. Inheritance Priority

i). SimulatorLite


TrafficVisLayer
-final double CAR_WIDTH
-final double CAR_PADD
-final double REL_WHEEL_WIDTH
-final double REL_WHEEL_LEN
-final VehicleStorage storage
~TrafficVisLayer(VehicleStorage storage)
+void paint(Graphics2D canvas)
~void paintCircle(Graphics2D canvas, Point3f p, int size)
-void paintCars(Graphics2D graphics)
+VisLayer create(VehicleStorage storage)
PedalController
-final float FORCE_CONSTANT
+void updateVehicleVelocity(Vehicle vehicle, long deltaTime, float steeringAngle, float pedalState)
Car
+Car(int id, int lane, Point3f position, Vector3f heading, float velocity)
+void update(long deltaTimeMs)
-void updateLane()
<<abstract>>
Vehicle
-final float MAX_STEERING
-int id
-int lane
-Point3f position
-float velocity
-Vector3f heading
-PlanController planController
-float axeLength
-float steeringAngle
-final VelocityController velocityController
#Vehicle(int id, int lane, Point3f position, Vector3f heading, float velocity)
+int getId()
+int getLane()
+void setLane(int lane)
+Point3f getPosition()
+void setPosition(Point3f position)
+float getVelocity()
+void setVelocity(float velocity)
+void scaleVelocity(float factor)
+void setVelocityVector(Vector3f velocity)
+Vector3f getHeading()
#void setHeading(Vector3f heading)
+float getAxeLength()
+float getSteeringAngle()
+void setSteeringAngle(float steeringAngle)
+PlanController getPlanController()
+VelocityController getVelocityController()
+void countSteeringAngle()
+void setWayPoints(Collection<Action> Actions)
+abstract void update(long deltaTime)
Ghost
+Ghost(int id, Point3f position)
+void update(long deltaTime)
<<implements>>
is
is
1..1
1..1
1..1
storage
1..1
1..1
1..1
ghosts
0..*
<<interface>>
ControllerInterface
+void updateVehicleVelocity(Vehicle vehicle, long deltaTime, float steeringAngle, float pedalState)
1..1
1..1
1..1
controller
1..1
1..1
1..1
0..*
vehicles
VehicleStorage
-Map<Integer, Vehicle> vehicles
-Map<Integer, Ghost> ghosts
-ControllerInterface controller
~long lastUpdate
+VehicleStorage(EventBasedEnvironment environment)
+void addVehicle(Vehicle vehicle)
+void addGhost(Ghost ghost)
+Vehicle getVehicle(Integer idVehicle)
+Ghost getGhost(int ghostID)
+void handleEvent(Event event)
+Collection<Vehicle> getVehicles()
+Collection<Ghost> getGhosts()
+LinkedList<Vehicle> getAllVehicles()
+RadarData generateRadarData()
NetVisLayer
-final NetLayer netLayer
+NetVisLayer()
+void paint(Graphics2D canvas)
+VisLayer create()
VelocityController
-final float DELTA_TIME
-final float MAX_ACCELERATION
-final float EPSILON
-float acceleration
-float desiredVelocity
-final Vehicle vehicle
+VelocityController(Vehicle vehicle)
+void updatePlan(Collection<Action> plan)
-void setAcceleration(float desiredVelocity)
+void updateVelocity(long deltaTime)
1..1
velocityController
1..1
vehicle
1..1
1..1
1..1
planController
1..1
1..1
PlanController
-LinkedList<Point3f> wayPoints
-final float SAFE_DIST_CONST
+PlanController()
+Point3f getNextWayPoint(Point3f pos, float velocity)
+LinkedList<Point3f> getWayPoints()
+List<Point3f> getNNextWayPoints(int n)
+List<Point3f> getAllNextWayPoints()
+void setWayPoints(LinkedList<Point3f> wayPoints)
+boolean isInMinDist(Point3f pos, float velocity)
1..1
1..1
1..1
1..1
vehicleStorage
mainVehicle
1..1
1..1
SimulatorCreator
-Simulation simulation
-SimulatorEnvironment environment
-final String CONFIG_FILE
-final Logger logger
+void init(String[] strings)
+void create()
-void createVisualization()
+void runSimulation()
+void main(String[] args)
storage
1..1
environment
1..1
SimulatorEnvironment
-final Logger logger
-VehicleStorage vehicleStorage
-Communicator communicator
+final long UPDATE_STEP
+final long COMM_STEP
+SimulatorEnvironment(EventProcessor eventProcessor)
-void initCommunication()
+void init()
-FactoryInterface createPlanFactory(String protocol)
-FactoryInterface createUpdateFactory(String protocol)
-Communicator createCommunicator(String protocol, String serverUri, boolean receiveThread, boolean sendThread)
+VehicleStorage getStorage()
+Communicator getCommunicator()
ZoomVehicleLayer
-final double MY_ZOOM
-final int PLACE_FOR_TEXT
-final int HEIGHT
-final int RADAR_HEIGHT
-final int WIDTH
-final int RADAR_WIDTH
-final int WIDTH_LINE
-final int TEXT_OFFSET
-final double SEARCHING_DIST
-final int VELOCITY_WIDTH
-final int VELOCITY_LENGTH
-final int CAR_WIDTH
-final int CAR_LENGTH
-final int WHEEL_WIDTH
-final int WHEEL_LENGTH
-final int LANE_WIDTH
-final VehicleStorage storage
-int indexOfCar
-Vehicle mainVehicle
-Rectangle2D drawingRectangle
-Network net
-boolean settingRotation
#ZoomVehicleLayer(VehicleStorage storage)
+void paint(Graphics2D canvas)
+VisLayer create(VehicleStorage storage)
+void init(Vis vis)
+String getLayerDescription()
-Vector2d vector(Vehicle main, Vehicle another)
-Vector2d vector(Vehicle main, Point2f p)
-boolean isIn(Vector2d v)
-boolean isIn(Point2d p)
-void drawVehicle(Graphics2D canvas, Vehicle vehicle)
~void paintSCALELine(Graphics2D canvas, Vector2d v1, Vector2d v2)
~void paintLine(Graphics2D canvas, Vector2d v1, Vector2d v2)
<<enumeration>>
SimulatorEvent
UPDATE
COMMUNICATION_UPDATE

ii) AgentDrive


HighwayEnvironmentHandler
+ addSensor(Class<C> clazz, Entity entity) : <C extends Action> C
+ addAction(Class<C> clazz, Entity entity) : <C extends Sensor> C
+ getEnvironmentDimensions() : Vector3d
Entity
+ Entity(String name)
+ getName() : String
DashBoardController
+ create()
+ getEventProcessor() : EventProcessor
+ handleEvent(Event event)
+ init(string[]args)
+ initTraffic()
+ runSimulation()
+ runSimulator(String name)
+saveDistance(Point2f p1, Point2f p2) : boolean
Sector
+ getFrom() : String
+ getLaneByIndex(int laneIdx) : Lane
+ getLanes() : HashMap<String, Lane>
+ getPriority() : int
+ getTo() : String
+ putLanes(HashMap<String, Lane> laneMap)
is
EventBasedHandler
+ eventProcessor : EventProcessor
+ handler : EventBasedhandler
Agent
+ Agent(int id)
+ addActuator(VehicleActuator actuator)
+ addSensor(VehicleSensor sensor)
+ addInitialPosition() : Point3f
+ addInitialVelocity() : Point3f
+ getNavigator() : RouteNavigator
Junction
+ Junction(String id, String type, Point2f center, ArrayList<String>intLanes, ArrayList<Point2f>shape, ArrayList<Request>requests)
DefaultCreator
+ create()
+ createVisualisation()
+ init(String[]args)
+ runSimulation()
Connection
+ getFrom() : String
+ getFromLane() : String
+ getTo() : String
+ getToLane() : String
HighwayEnvironment
+ getCommunicator() : Communicator
+ getRoadNetwork() : Network
+ getStorage() : HighwayStorage
+ handler() : HighwayEnvironmentHandler
+ initProtoCommunicator()
cz.agents.highway.environment
+ HIghwayEnvironment
- RandomProvider
MotionAgent
+ Attribute 1 : Type
+ Attribute 2 : Type
- Attribute 3 : Type
- Attribute 4 : Type
+ Operation 1 (arg list) : return
+ Operation 2 (arg list) : return
+ Operation 3 (arg list) : return
+ Operation 4 (arg list) : return
Edge
+ Edge(String id, String from, String to, String priority, String type, ArrayList<Point2f>shape)
+ getFrom() : String
+ getLaneByIndex(int laneIdx) : Lane
+ getLanes() : HashMap<String, Lane>
+ getPriority() : int
+ getTo() : String
+ putLanes(HashMap <String, Lane>)laneMap
EventBasedEnvironment
+ EventBasedEnvironment(EventProcessor eventProcessor)
+ getEventProcessor() : EventProcessor
+ handler() : EventBasedHandler
<<Interface>>
FactoryInterface
+ doMethod (String) : void
+ init (boolean, long) : boolean
CarManeuver
+ CarManeuver(CarManeuver pred)
+ CarManeuver(int laneIn, double velocityIn double positionIn, long startTime)
+ equals(CarManeuver otherCarManeuver) : boolean
+ getAccelertaion() : double
+ getDistanceInFirstLane() : double
+ getDuration() : double
+ getEndTime() : long
+ getExpectedLaneOut() : int
+ getGuidePointsCount() : int
+ getLaneIn() : int
+ getLaneOut() : int
+ getLength() : double
+ getPositionIn() : double
+ getPositionOut() : double
+getStartTime() : long
+ getvelocityIn() : double
+ getVelocityOut() : double
+ isFinishing() : boolean
+ isInfiniteInTime : boolean
+ isSafeManeuver() : boolean
+ setFinishing(boolean b)
+ setGuidePointsCount(int guidePointsCount)
+ setInfiniteInTime(boolean isInfiniteTime)
+ setSafeManeuver(boolean IsSafeManeuver)
+ toString() : String
PlansFactory
+ decode(Message pm) : PlansOut
+ encode(Object object) : Message
+ getObjectClass() : Class<PlansOut>
LocalSimulatorHandler
+ LocalSimulatorHandler(ProtobufFactory f, Set<Integer>plannedVehicles)
+ executePlans(PlansOut plans)
+ sendPlans(Map<Integer, RoadObject> vehicleStates)
InitMessageFactory
+ decode(Message pm) : InitIn
+ encode(Object object) : Message
+ getObjectClass() : Class<InitIn>
SimulatorHandler--
+ addActions(int id, List<Action> actions)
+ hasvehicle(int vehicleId) : boolean
+ isReady() : boolean
+ numberOfVehicles() : int
+ sendPlans(Map<Integer, RoadObject> vehicleStates
UpdateFactory
+ decode(Message pm) : CarsIn
+ encode(Object object) : Message
+ getObjectClass() : Class<CarsIn>
AccelerationManeuver
AccelertaionManeuver(int laneIn, double velocityIn, double positionIn, long startTime)
AccelertaionManeuver(CarManeuver pred)
StraightManeuver
+ StraightManeuver(CarManeuver pred)
+ StraightManeuver(int laneIn, double velocityIn, double positionIn, long startTime)
LaneLeftManeuver
+ LaneLeftManeuver(int laneIn, double velocityIn, double positionIn, long startTime)
+ LaneLeftManeuver(CarManeuver pred)
DeaccelerationManeuver
+ DeccelertaionManeuver(int laneIn, double velocityIn, double positionOut, double positionIn, long startTime)
+ DeccelertaionManeuver(int laneIn, double velocityIn, double positionIn, long startTime)
+ DeaccelertaionManeuver(CarManeuver pred)
LaneRightManeuver
+ LaneRightManeuver(int laneIn, double velocityIn, double positionIn, long startTime)
[online diagramming & design] creately.com

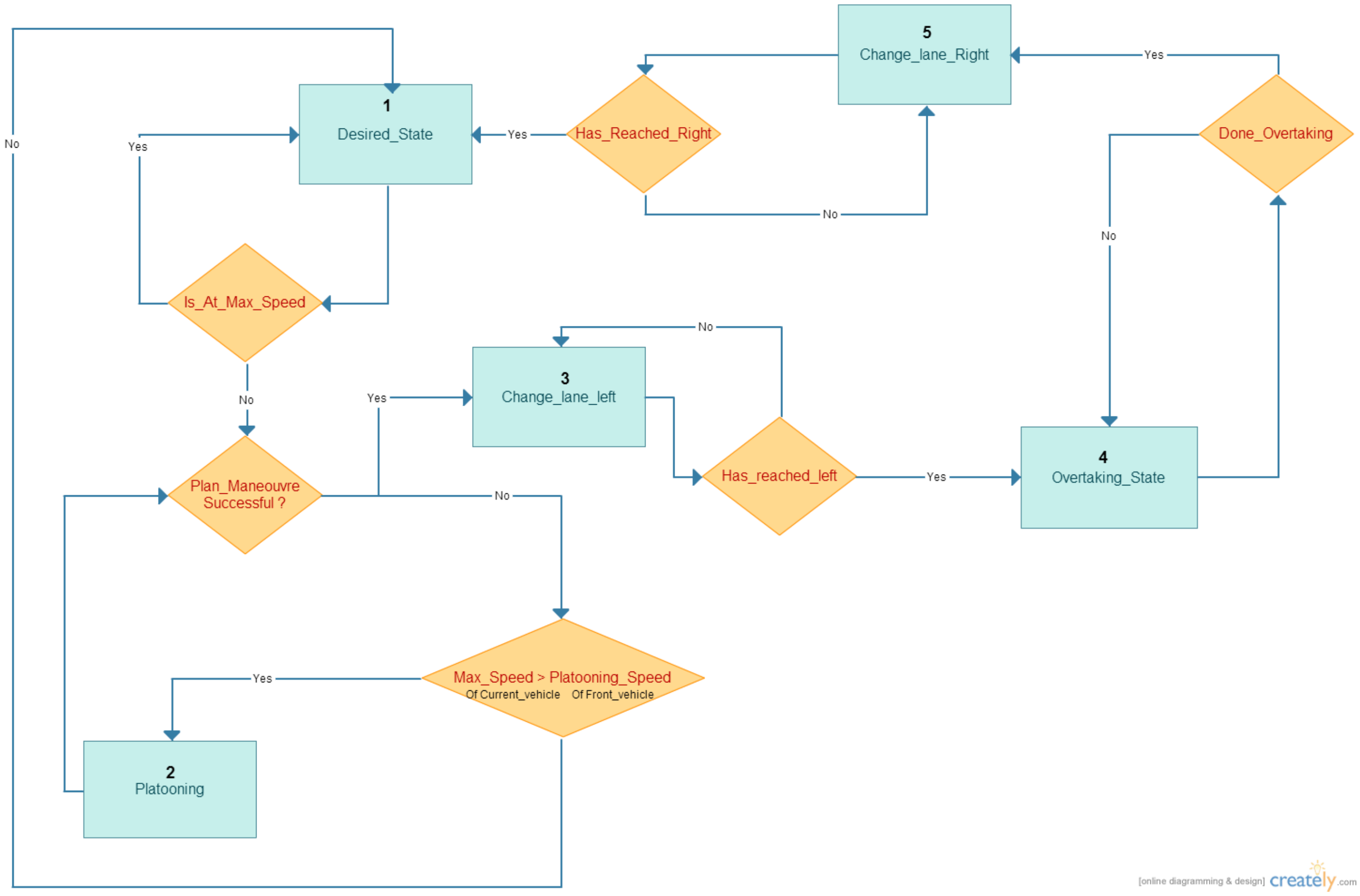
3. Tutorial on Autonomous Driving

a. Description:

The goal is to provide a waypoint to an agent. A Waypoint acts like a guide to the agent (or a vehicle) and the vehicle tries to follow that Waypoint. It can be done in the following two ways:

1. There are several types of maneuvering possible in AgentDrive like Acceleration maneuver, Straight maneuver, Deacceleration maneuver and so on. So, manipulating waypoints for an agent in different scenarios (say while overtaking) will work. For this, We can instantiate an object of StraightManeuver Class with parameters like current lane, velocityIn, laneout etc. and further an object of Maneuvertranslator class can call its member function "translate", which takes an object of StraightManeuver as a parameter, to give us the next Waypoint for the agent.
Thus by changing the reference to the type of maneuver desired, we can obtain desired Waypoints in sequential order for an agent.

2. An another approach to deal with it will be defining the Waypoints in each scenario, considering any kind of translation and pre-defined Maneuvers. We could develop a State cycle (as shown below), which will uniquely define a Vectorial change in Waypoint for each state. These states loop among themselves to achieve autonomous overtake.



> *Diagram Description:*

1. *An agent always tries to reach its defined maximum speed and at the same time senses the vehicle in front of it and thereby maintains itself in desired State.*

2. *If it crosses the safe_limit(say 40metres) and also it is not travelling at it's maximum speed then it either tries to plan a maneouvre or change its state to Platooning.*

3. *In platooning State it repeatedly tries to plan a maneuever else follow the vehicle ahead of it.*

4. *If a plan is found by the method Plan_maneouvre, the vehicle changes its state to Change_left_Lane and continuously loops until it has reached left lane.*

5. *After it has made to change its lane to left, it follows with a certain calculated speed (based on vehicle ahead of it in changed lane) and tries to loop until the parameters to remain in left lane are not violated.*

6. *After having reached the time_limit to stay in left lane it changes its state to change_right_lane and loops until it has changed to the right lane.*

b. How did we interface with AgentDrive:

Following are the instructions to implement second approach:

1. Create a new java class in AgentDrive/Source packages/cz.agents.highway.agent and inherit the class "Agent" to have features like adding_sensors, adding_actuators, get_inital_velocity and so on. These functions enable us to sense the environment around the agent.

2. Since "agentReact" method is called for every agent after each timestep, we need to return the updated vector to move the agent in the desired direction. And the Waypoints are decided according to the states of an agent.

3. We also need to define a method say, Plan_manoeuvre which plans for a possible manoeuvre in advance. The physics involved in the whole process is:

   i) Sense and predict positions of rear_left and front_left after time

   t1 = (Distance_between_two_lanes/Maximum_speed_of_Agent).

   ii) If there is no safe distance to change state to left lane exit
   else compute overtake speed based on vehicle in front_left and predict time to stay in left lane:

   t2 =(Distance between vehicle to manoeuvre and Predicted position in left lane)+safe_distance(say 30)/(Relative speed between the vehicles)

   iii) Now, Predict the position as in step i).

4. If the Plan is successful return the Waypoints else return 0. Further, change states according to the returned parameters.