Basic Documentation of Code Structure

1. How to run the model

There are four main methods which are used to call the code:

- Random Simulation without UI: this has no user interface and will run a coded series of sets of random runs. Multiple different internal seeds can be used with each external seed, multiple external seeds can be used with each level of seeded faults, and this configuration can be repeated multiple times – this loop structure means that some outputs can be available for each fault level early in a batch run, with further data for statistical purposes coming later.
- Simulation with UI: launches the UI and allows runs to be initialised by the user. Without intervention, each run will be different; to re-run the same model, ensure that increment seed at end of run is unchecked (internal seed – Console tab), and copy and re-paste the external seed (Model tab) prior to pressing play.
- Run Specific Batch from File: uses a predefined list (from file) of internal and external random seeds, and constructs the network map, runs pre-processing and can return without running the simulations – would be easy to modify to run the simulations, or to return different pre-processing metrics.
- Run Comparison: this is used to run the main experiments and relies on a number of other methods, e.g. SearchBasedMapGeneration.generateExternalSeeds, ActuallyRunSpecificBatchFromFile.runBatch, COModelWithoutUI.runBatch, and CalculateSituationCoverageFromFile.calcSitCov. These have all ended up in separate classes as they were designed as independent 'main' methods for testing, but then were integrated into a single main method. This method allows generation of a map set by a search based method, simulation of this map set with specified, or random, seeded faults, and then for maps to be generated at random and simulated for an amount of time equivalent to that taken by the search based method. The situation coverage of each set of maps is also calculated and output. Various parts of the method can be activated or deactivated to produce variations on this experiment.

And two additional methods to analyse or simulate outputs further:

- Run Compare Map Sets: simulates two map sets in full, one after the other, against a defined or random fault array. Records the start and end time for each map set.
- Run Compare Map Set Coverage Dist: calculates the distribution of the situation coverage for two supplied map sets. This is the number of maps in the set which fall into each situation coverage category. Relies on CalculateSituationCoverageFromFile.calcSitCovDist method.

2. What are the 'agents' in our simulation?

There are a few main 'agents' in the simulation:

- **UGV** – single autonomous vehicle. Controller behaviour is meant to simulate 'real' autonomous vehicle i.e. vehicle uses sensors to locate itself in environment by searching for

road markings. It is permitted to query 'intelligent junctions' (see below) to ascertain whether it is approaching a junction (and should slow down), or to work out which exit to choose. It knows the location of its target and the goal is to reach the target. The vehicle is programmed to slow to a stop if it detects dangerous conditions e.g. approaching obstacle, and we assume that if the UGV is stationary, any collisions that occur are not the fault of the UGV

- **DumbCar** – moving (car) obstacles controlled by (some) non-autonomous methods i.e. the controller is permitted to use methods which would be considered 'cheating' for the autonomous vehicle e.g. findImminentCrash method which directly accesses the locations of all other vehicles in the model to check whether they are within about 20m of the DumbCar, then uses other calculations to find the closest one that is (probably) in the lane in front. These cars are not as sophisticated, and to avoid adding to UGV complexity, they do not perform overtaking manoeuvres, and are permitted to drive over other objects without failures being reported.
- **ParkedCar** – static (car) obstacles at a fixed offset from the kerb. These cannot be placed in junctions, junction approaches or junction exits. Obstacles may not overlap one another when in the same lane, and must be separated by an x AND y separation of less than Constants.OBSTACLE_LENGTH when calculated as absolute displacement in each direction (if the locations are in the same lane) or less than (Constants.OBSTACLE_LENGTH + Constants.OBSTACLE_BUFFER) * 1.5 (if the locations are in different lanes on the same road).
- **Junction** – 'intelligent' junctions attempt to prevent multiple vehicles from entering a junction at the same time. Provides methods for selecting an appropriate exit from the junction for the UGV (i.e. to take it closer to the Target, or to explore all exits), and a random exit from a junction for a DumbCar.

3. Which methods set up the network map and simulation?

The main classes and methods for controlling the set-up for the run are:

- COModelWithoutRun: this creates a new COModelBuilder using a passed in internal seed and percentage of faults, and during the start() method initialises the external seed (again passed in), giving complete control over the network that is being built. Once the network is built, it can be interrogated for environmental features which can be returned as a string from the start() method.
- COModelWithoutUI: this creates a new COModelBuilder using the system clock to set the internal seed, and a passed in percentage of faults. The simulation can be run as a batch, all with the same external seed, or individually by calling start() directly. In both modes, the network map is generated for each run, and then the simulation is carried out on that network with results being returned to log/results files.
- COModelWithUI: this class constructs the model within a GUIState, so the network is only built when the simulation is started from the interface. If the external seed has just been set in the interface then it is used, otherwise a new seed is generated at random. Various different visual representations are set up to portray the running simulation, and it is shown in the MASON GUI; the model can be paused, stopped and restarted. If the same run is

required to be repeated, the internal seed tick box should be cleared before the end of the run so that it is not incremented, and the external seed box should be copied and re-pasted over the top immediately prior to the new run. NOTE: the map visualisations do not scale well using the MASON GUI, this is probably due to incomplete implementation in the setupPortrayals method. ALSO: it is not currently possible to perform video capture on the simulation, however this may be an issue with the configuration/drivers on my local machine, rather than an issue with the code.

4. Which methods run the simulation?

The simulation is controlled from:

- COModel.start: this method sets up the model by loading and scheduling all the dynamic entities (anything with a step method that needs to be executed on each iteration), including the AccidentDetector. It alsobuilds all the mappings for the static objects such as the Junctions and Roads.

The simulation ends when either:

- The AccidentDetector detects that the number of iterations has exceeded the maximum of 5000; or
- The UGV.step method calls COModel.dealWithTermination after the UGV has reached the Target and is no longer active. This method detects that there are no longer any active UGVs and therefore it kills the simulation.

5. Which methods provide fault detection, logging and outputs?

The AccidentDetector class is primarily responsible for fault detection, logging and outputs, although it is assisted by methods implemented in COModel; an additional InfoLogFile class provides a longer log for information purposes, although this is overwritten each time a new simulation batch is started. Two Accident log files are produced, a log (detailed message for each accident that occurs, along with header and footer information, including a list of which seeded faults were active, and the number of times each of them were called), and a summary (single line per run, with information about the network configuration, and totals on the types of accidents etc.) The configuration of the run determines how the accident log files will be named, and whether they will contain multiple sets of results. These files can be easily overwritten if the run is repeated with the same configuration, so careful consideration must be given to the way that the files are named, or alternatively, the destination folder (outFilePath) should be changed in the Constants file before each new run. In general, if the method for calling the model creates a new COModel , then a new AccidentDetector is created, and this will create new log files. Where a single COModel instance is used to run multiple simulations e.g. in RandomSimulationWithoutUI which loops for 5 different maps, and executes runBatch to give 3 different simulations on each map, there will be 15 sets of outputs in each log file. Note that this method also contains two outer loops, one which sets the percentage of faults to inject, and another which repeats the experiment 20 times. A new output file will be generated for each level of percentage faults, and for each outer loop. Note that the percentageFaults, and outer

loop index are provided to the COModelWithoutUI constructor in order that they can be used in the naming of the output files to ensure that files are not overwritten during the experiment. If the experiment were repeated a second time without changing the destination folder, these files would be overwritten. For the format of the output filenames, see the AccidentDetector constructor.

The AccidentDetector has a step method which runs on each iteration in order to check to see if the network has entered an accident state. It does this by making the following checks for the UGV:

- CLASHWITHOBSTACLE (Parked Car) – Iterates through all obstacles and runs obstacle.inShape(ugv.getShape())
- LEAVEROAD – Iterates through all roads and runs road.inShape(car.location) on each. If the car location is not detected on any roads then true will be returned. NOTE: This only checks to see if the centre of the UGV has left the road (will assume this is good enough for now, as many slight over-lapping incidents with the pavement would cause a lot more accidents to be recorded).
- CROSSCENTRELINE – Tries to work out whether (and where) the UGV has crossed the centre line by 'plotting' a line between previous and current location and checking for intersection with the centre line shape on each road (Rectangle2D.intersectsLine). Line crossings that occur when the UGV is overtaking, U-turning, or when the 'collision' point occurs in a junction, are ignored. NOTE: It may be possible to return two collisions (on adjacent timesteps) for a single intersection if the UGV ends the first step 'on' the line (i.e. within the boundary of the 10cm wide line).
- CROSS_NE_LINE – As for CROSSCENTRELINE, but checks against the NB or EB lines for each road.
- CROSS_SW_LINE – As for CROSSCENTRELINE, but checks against the SB or EB lines for each road.
- CLASHWITHOTHERCAR – Iterate through the bag of cars and work out whether the UGV has crashed into any of the *other* cars. NOTE: If the UGV is stationary at the time of the collision, then it is deemed not to be at fault, and no accident is recorded. Method creates two areas using ugv.getShape() and car.getShape() and then creates a third area which is the intersection of the other two. If this area is non-empty, then a collision has occurred, and we must determine whether the UGV is at fault, making it a valid collision to report as an accident/failure. Situations which are not deemed to be the fault of the UGV:
    - If the UGV is not in a junction, and the DumbCar is mostly located in the other lane (we'll just check which side of the road the centre of each car is on) then we can assume that the collision is the fault of the DumbCar. Alternatively, as long as the UGV is on the side of the road corresponding to its direction of travel, if it is hit by a vehicle travelling in the other direction, assume the DC is at fault.
    - If the DumbCar is travelling faster than the UGV, on approximately the same bearing (+/- 45 deg) then we can assume that the collision is the fault of the DumbCar - hopefully this will eliminate the failure of a DumbCar driving over the top of a UGV that is slowing down to a stop, but has not quite stopped yet.

- If the DumbCar is spanning two lanes i.e. intersecting with the centreline, then we can assume it is at fault in the crash.

It also checks to see if the model has exceeded the maximum of 5000 steps, as this represents a Timeout Failure (which is also counted as an Accident).

6. How does overtaking work?

Overtaking only applies to the UGV class, and the ParkedCar (or Obstacle) class that it is overtaking as the DumbCars in the simulation are not able to overtake.

- The UGV uses its sensors to detect nearby static obstacles, and records the coordinates of the closest and furthest points found; it also looks for moving obstacles in the opposite lane (i.e. oncoming cars).
- For a UGV that is NOT in overtaking mode, if an obstacles is found within viewing range (25m), we need to check whether it is safe to overtake:
  - Assuming that the oncoming vehicle is travelling at maximum speed, determine whether there is time (in simulation steps) for the UGV to travel the distance to, around and past the obstacle (assuming that the UGV is starting from its current speed and accelerating to maximum speed in order to complete the manoeuvre) before the oncoming car reaches the point where the UGV has pulled back in.
    - Begin the manoeuvre when the UGV will be within the OBSTACLE_HEADWAY distance (7m) on the next step (based on current speed).
    - Create a waypoint (TPARKEDCAR) that is slightly ahead of (2m) the end of the obstacle, and is offset in the lane by the current lane position + the OBSTACLE_WIDTH distance (2m) + 0.5m. [This should result in the UGV passing the obstacle with a clearance of 1m as the centre of the UGV should track at an offset of 4m from the kerb (spanning 3m-5m) while the obstacle should span 0m-2m. NOTE: This does not necessarily happen in practice due to the way the UGV is determined to have reached the target (see later). The vehicle enters stage OVERTAKE_START.
    - This code uses crude grid-based geometry and would need updating if the network map is upgraded to allow roads at arbitrary angles.
  - If there is not time for an overtake before the oncoming vehicle approaches:
    - See if the UGV is going to encroach on the OBSTACLE_HEADWAY distance (7m) on the next step, and if so, execute an emergencyStop() – this is essentially a cheat where the speed is set to 0 and the method returns the speed overshoot (speed – max deceleration) so that this can be logged (although this is only done in the infoLog, not classed as an Accident/Failure).
    - If the vehicle is a little further away than this, then slow down (uses goReallySlow() rather than goSlowStop, so at the moment does not force a complete stop).
    - For both the above situations, enter stage WAIT.
- For a UGV that is in overtaking mode (i.e. waypoint type = TPARKEDCAR):

- Is it close to the waypoint – defined as within 1m, or half a step (whichever is greater) – to ensure that we can't miss the waypoint completely. OR has it overshot (defined as having passed it in the compass direction closest to the UGV's direction of travel) the waypoint. If so:
    - If the UGV is approaching the first waypoint (stage = OVERTAKE_START) then we need to know whether this is the only obstacle that we can see, or if there is another obstacle after this one.
        - If this is the only obstacle, or we think there might be a few obstacles in a row, create a new waypoint that is in line with the current location of the UGV and the offset slightly further than the furthest point on the obstacle. [The problem with this is that the UGV could be 1.25m away from the original waypoint when this code calls, so the new waypoint could be added 1.25m closer to the obstacle, and as we only have clearance of 1m, this results in a collision.] .
            - If we think that there might be an obstacle overlapping the limit of our range (such that the WP we just added wasn't inserted far enough away, then we keep the state as OVERTAKE_START, so we can add a new one at the right distance when we get there.
            - Otherwise, new state = OVERTAKE_PULLEDOUT.
        - NOTE: there is a seeded fault here that, when there are likely to be 2 obstacles, we are going to insert another waypoint as if it was the first one again (which means that we offset from the current location by the OBSTACLE_WIDTH + 1m, and therefore this will result in a divergence from the intended path).
        - There is a final conditional clause which is intended to be executed if there is a sufficient gap between the vehicle being overtaken, and the next obstacle detected, such that the vehicle should pull back in, between the two parked cars. This code is unlikely to be executed as it requires the max distance to the obstacle to exceed the range of the sensor; something which shouldn't occur (except in situations where the crow-flies distance is marginally longer than that used for the sensor range. Either way, this code does not function as required.
    - If the UGV is in OVERTAKE_PULLEDOUT, or has somehow passed the end of the obstacle without leaving OVERTAKE_START, we can create the final waypoint for returning the vehicle back to the normal driving position. The waypoint is created at a lane offset relative to the current position of the UGV, displaced back into the lane by OBSTACLE_WIDTH distance (2m) + 0.5m, and away from the current location by the OBSTACLE_HEADWAY further down the lane. NOTE: if the UGV has 'reached' the waypoint from a position which is close to 1.25m away from the waypoint then the new waypoint may be added either too close to the kerb, or too close to the centre line, and the trajectory might take the vehicle too close to the

obstacle. As mentioned previously, this overtaking code should be revised so that it builds up a virtual picture of the obstacles it is overtaking, so that it knows where there are gaps which are big enough to pull into. It should also use the road markings to work out what the lane position should be, rather than guessing relative to current location (and constant obstacle widths) much more detail should be put into storing/analysing historical sensor data to build a more accurate picture of the local environment.

        - If the UGV is in OVERTAKE_FINISH mode and has reached the waypoint then it should eat the WP and get back the 'real' target. Mode is set to NOT_OVERTAKING. NOTE: suspect this code does not actually work as the intermediate waypoints are not removed before the new ones are added, so the actual target waypoint is probably orphaned. There is code further on that will reset the eTarget to the finalTarget when it has been set to an invalid waypoint, but this code could be tidied up so that the failsafe is not necessary.
    - If the UGV is not close to the waypoint, then make sure it keeps turning towards it during this step. There is a seeded fault which causes the UGV to forget to do this, which can cause the trajectory to continue in a straight line, and therefore might miss the waypoint altogether (although because of the range at which waypoints can be 'collected', instead we might just see the UGV in a less than optimal location when it collects the waypoint; this mistake can propagate due to the way that some of the future waypoints are set relative to the current location of the UGV.

7. What behaviour do the other cars (DumbCar) have?

A randomly selected number of these cars (up to 20, ideally with a minimum of 5, but this may not be possible if the road network is small or there are a lot of obstacles) are added at random locations on the road network. Cars may not be added in junctions, where there is a static obstacle (ParkedCar) on the road, or to be within 5m of the start location of the UGV (this is centre to centre distance, so is actually just a 1.5m gap if vehicles are in the same lane).

- If the car has left the road, remove it from the simulation (releasing any locks on junctions) and adding a new car at one of the entrances to the network (so that the number of cars in the simulation remains constant).
- Cars use waypoints to control turning at junctions, but otherwise do not have any specific target to achieve, nor do they take any action to avoid obstacles by using waypoints to execute an overtaking manoeuvre. At the start of each step, any existing waypoint information will be retrieved.
- If the Car has just entered a junction, an exit from the junction is chosen at random – this can include u-turns and a vehicle is permitted to leave the network at a dead-end (but only by continuing in its current direction of travel). A waypoint is set just outside the junction in the chosen exit lane and the vehicle makes a request to slow down. If the junction is occupied, or if there is a timeout trying to choose an exit, the car will execute an emergency stop (which may involve it breaking some laws of motion, or at least exceeding the vehicle performance spec) and start waiting. If the car is inside a junction approach, it should make

a request to slow down (goReallySlow). If the car is already inside a junction, it should continue to slow down so that it turns as tightly as possible. NOTE: 1. It is possible that the turn speed is slower than it needs to be as some vehicles really crawl through the junction. 2. This code assumes a grid-based network.

- If the Car is turning, the waypoint can be cleared when the Car leaves the junction (or enters another adjacent junction); the junction lock should also be cleared. If the Car is still in the junction, keep adjusting the direction to point at the waypoint.
- For collision avoidance, the Car uses a simple method of comparing its location to that of all items in a 'Bag', using perfect knowledge (since we are not interested in how realistic our Car simulation is – these are just 'human drivers' and so it's okay to 'cheat' with the 'sensing'. The Car works out its nearest obstacle (UGV or other car) by finding obstacles within a radius of 25m and checking to see whether they are in front of the vehicle or behind it (returns the shortest distance and +/- to indicate direction). NOTE: This assumes a grid-based network. This vehicle is set as the 'stopping distance' and compared to the previous stopping distance to ascertain whether the vehicle is getting closer to the obstacle (in front) and a slowdown should be requested. Otherwise if the vehicle behind is closer than 10m, and the Car is not in waiting mode, then it should be forced to speed up as the vehicles are getting too close. Otherwise, the Car should try to speed up as there may be no reason not to.
- If the vehicle is not in a junction, and is not waiting, and is not about to enter a junction (as determined by working out, using snap to lane (to make sure we are still travelling at the right lane offset etc.) where we should be on the next step if we continue at the current bearing, and on max speed), then adjust the bearing to point towards this location.
- If we are not in a junction or approach, and are not waiting, put in a vote for a speed up, then do speed calcs to resolve all the requests that have been made, and work out the x and y displacements necessary to effect the movement at the current speed. Update the Car location as required.

8. Dominant

Classes with Main Methods

- RandomSimulationWithoutUI
    - Run an outer loop (length specified by Constants.NO_RANDOM_RUNS) which controls an inner loop (iterating through different percentages of seeded faults) which constructs a simulation with the appropriate level of seeded faults. For each of these simulations, 5 different maps are generated (controlled by ExternalSeed, a randomly generated random number seed) and these are run with 3 different sets of internal behaviour (controlled by an internal random seed). On each occasion, the seeded faults are set independently, so although they are at the same level, different faults will be active on each run.
- RunComparison
    - Perform Search Based map set generation to generate a set of external random seeds which each define a map. Search is limited by the iterationLimit, which is the number of candidate maps that the search is allowed to evaluation for its set.

    - Run each map in the chosen map set (from the list in the output file produced by the previous step). Depending on the exact configuration coded, you can set a percentage of faults to be active at random on each run, or supply a list of faults which should be activated one-at-a-time and tested against the map. Each of these tests can be repeated once, or multiple times for statistical confidence.
    - The time to complete the two steps above is used as a benchmark for the effort that a Random map generation strategy can use to produce and test maps; a new map can be generated as long as the allocated time has not been exceeded, all simulations required for that map can then be completed, even if the time limit is exceeded at some point during those simulations: "I've started so I'll finish…". As for the Search Based map set, a fixed percentage of faults can be activated at random for each run, or faults from a specific list can each be tested against every map. Each of these experiments can be run once, or as a batch with different internal random seeds for statistical purposes. The strange numbers that are supplied to the COModelWithoutUI constructor are used to assist in the naming of output files so that other results are not overwritten. The External Seed that defines each map that is tested is stored in an output file: RandomExternalSeeds_*.txt, and when the time limit has been exceeded and the randomly generated map set is complete, the situation coverage of this map set is computed.
    - Output: RunComparisonLog.txt - Iteration limit, start time for Search Based map generation, situation coverage achieved by corresponding map set, start time for Random map generation, end time for random map generation and situation coverage achieved by corresponding map set.
- RunSpecificBatchFromFile
    - This method reads the following parameters from a text file: Internal Seed, External Seed, and Percentage Faults – they appear in a comma separated list, one set per line, terminated with a final comma. These parameters are used to construct the map which is then interrogated for certain a-priori (proposed) coverage measures which are then returned as a string for printing to the output file. NOTE: This method does not actually run the simulation, it merely gets the simulation ready for a run, and can be used to translate the external random number seed which defines a map into the geometric properties of that map which might help in the calculation of situation coverage.
- SimulationWithUI
    - This launches the UI and allows runs to be initialised by the user. Without intervention, each run will be different; to re-run the same model, ensure that "Increment Seed on Stop" box is unchecked (internal random number seed) on the Console tab, and copy and re-paste the external seed (on the Model tab) prior to pressing play.
    - To repeat a run from a log file, paste the Internal Seed from the log file into "Random Number Seed" on the Console tab, and the External Seed from the log file into "ExternalSeed" on the Model tab. If you want to repeat the run multiple times, see note above.
    - The Delay slider on the Console tab can be used to slow down the model. There are also options on this tab to insert stops/pauses under various conditions.

Pause/Stop/Play buttons are also available to interact with a running model. It should be possible to record video, and or take snapshots of the model using the Environment Display, although I have not managed to get these to work on my machine (requires installation of JMF?).

Helper Classes

- ActuallyRunSpecificBatchFromFile.runBatch
    - Reads External Random Seeds from file (selectedExternalSeeds_*.txt) and runs the map generated by each against either a randomly selected subset of active faults (percentage active determined by local variable: percentageFaults), or against each fault (individually) in a supplied set of faults. The method can be configured to run each map/fault combination once, or repeat multiple times with different internal random seeds (these will result in different random choice behaviour e.g. at junctions, and car obstacles being added in different locations).
    - This method is called *Actually* Run Specific Batch from File because it does actually run the simulation, allowing the UGV to move around and attempt to reach the target (in contrast with RunSpecificBatchFromFile which only sets up the map, see above).
- CalculateSituationCoverageFromFile.calcSitCov
    - This method is used to calculate the Situation Coverage achieved by the Random map set once all the random seeds have been added to the map set file.
    - Create a 3D array to represent the 3 dimensions of the Situation Coverage metric, containing 'boxes' for each of the categories that each metric is divided into. Create a map based on the External Seed extracted from the map set file, and calculate the values of the three metrics, working out which category each one falls into, and ultimately which box in the 3D array should this map would fall into.
    - If the 'box' contains less than the required number of maps for it to be 'covered' (in the case of our experiments, this was set to 1), increment the count in the box. If we have now reached the required count, increment the count of the number of boxes which are covered.
    - Repeat for all maps in the maps set, and once complete calculate the percentage Situation Coverage achieved. This is written to an output file, along with a series of 2D arrays which represent the 3D array.
- CalculateSituationCoverageFromFile.calcSitCovDist
    - This is similar to the above, except it can be run for both Random or Search Based map sets (input parameter), and instead of returning the percentage Situation Coverage, or being limited by the required count in each box, this method just counts the total number of maps which fall into each box in the 3D array. This gives us a measure of the distribution of the map set across the Situation space that we have defined.
    - The output is a series of 2D matrices dumped to file to represent the 3D array.
- SearchBasedMapGeneration.generateExternalSeeds
    - Similar to CalculateSituationCoverageFromFile.calcSitCov (above) in that the map is constructed, metrics taken and the appropriate 'box' in the 3D matrix is checked to

see if it has reached the required coverage. If it has not, then the External Seed is added to an the output file containing the map set. This search is limited by the input parameter interationLimit, which constrains the number of random maps that we can generate and test to see if they improve our map set. The search will also complete if 100% Situation Coverage is achieved.

- This method also produces an output file containing the Situation Coverage information, and the 2D matrices to represent the 3D array.

Not used (should be removed when code is tidied up)- Note: I have attempted to remove these from the svn repository, but I must be doing something wrong as they keep reappearing when I check the code out.

- Simulation
  - This appears to control evolutionary search (deprecated).
- OurMutatorPipeline
  - Method used by Simulation for evolutionary search (deprecated).
- MaxAccidents
  - Method used by Simulation for evolutionary search (deprecated).
- Car-obstacle.params
  - This is a parameter file used during the evolutionary search (deprecated).

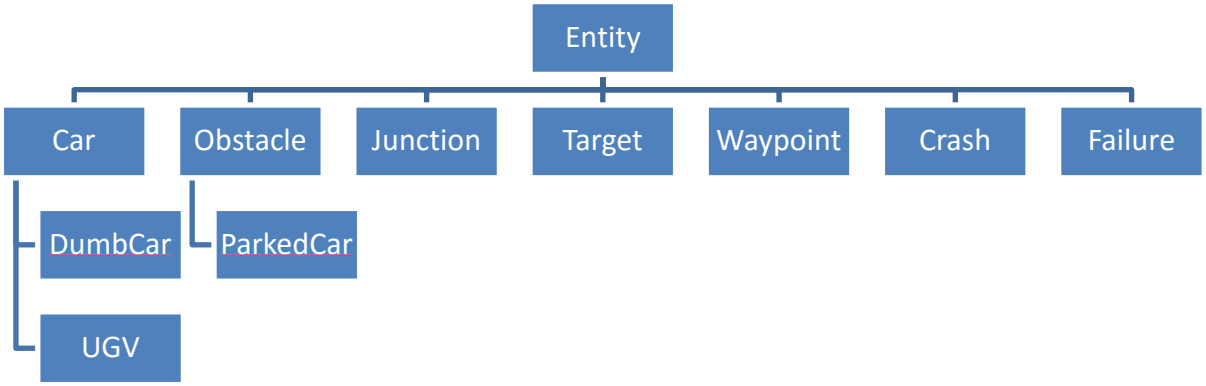
Modelling

Classes relating to simulation/map setup

- COModel
- COModelBuilder
- COModelWithoutRun
- COModelWithoutUI
- COModelWithUI

Classes for 'real' entities/objects

- Car
- Crash
- DumbCar
- Entity
- Failure
- Junction
- Obstacle
- ParkedCar
- Road
- Target
- UGV
- Waypoint



Utility Classes

- CarPerformance
    - This class mainly exists to facilitate altering the performance of the vehicle under different terrains/driving conditions e.g. rain/snow/ice/grass/sand/gravel. These conditions are not simulated in the current model, but the code has been retained to support future extensions.
- Constants
- Utility

Logging/Output Classes

- AccidentDetector
- InfoLogFile