# Part C - Driving It Home

SWEN30006, Semester 1 2016

### Overview

After seeing your work on improving the Metro Melbourne simulation, you have been contacted by the City of Melbourne who are partnering with an undisclosed car manufacturer to help turn Melbourne into a truly 21st century city by introducing fully self driving cars into the Australian market.

So far, the City of Melbourne has developed a simplified simulation of a road network, including traffic lights, signage and buildings. The simulation currently allows manual driving of cars, but nothing further. Working as a team, your task is to design and implement one of three subsystems required to build and run the self-driving car. The City of Melbourne will be combining your work with another two contracted teams to realise the full self driving car experience, and test different combinations within the simulated environment.

## The Simulated World

The City of Melbourne has done the preliminary design work on building a basic world simulation in which you can test your self driving car. This world is currently very simple, consisting only of:

- Roads
- Traffic Lights
- Signs
- Street Lamps
- Buildings

The roads themselves form a directed graph where intersections are the nodes, and currently only support 90 degree corners within the road. This kind of simulation is well suited to modelling inner city driving, where grid like structures are prevalent. Currently the simulation supports multiple types of intersections, and multi lane roads by associating road markings with road sections

This system is modelled in the design class diagram in Figure 1.

The City of Melbourne plans to add complexity to the simulation in the middle of next year, something you may want to consider when designing your subsystem. Currently this system operates on a standard simulation loop, first updating all objects, then rendering them, continuing until the application is closed. The simulation world also includes a keyboard based controller implementation of the `IControl` interface, allowing you to drive the sample car around the environment using the `W`,`A`,`S` and `D` keys. *You should ensure that you can build and run the system provided to you before beginning any implementation.*

### Gradle

The provided simulation makes use of Gradle as a build and dependency management tool. By using Gradle, there is no need to manually link any of the libraries or dependencies within the project, and it is much easier to migrate between development environments and IDEs. In order to use Gradle within Eclipse you must do two things:

- Install the Eclipse Gradle integration by following the installation guide.
- Import the Gradle project into Eclipse using the LibGDX Guide.

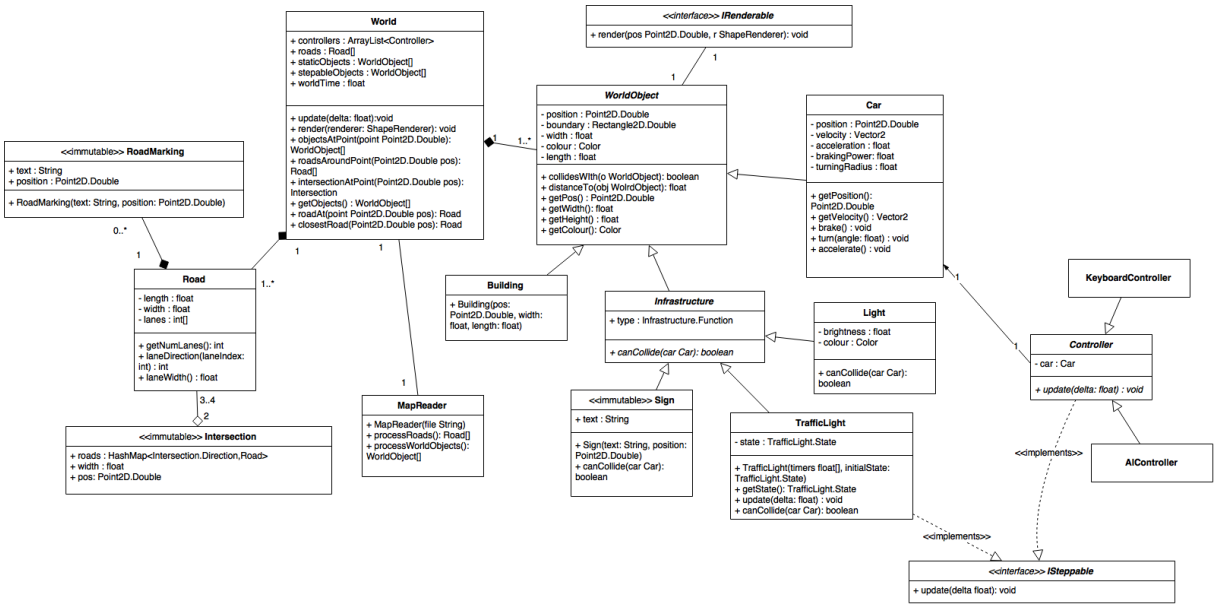


Figure 1: System Design Class Diagram

If you have already completed the installation of the gradle integration for Part B you will not need to do it again.

Your first implementation task should be to import the project, and build and run the application, to ensure that you have the existing simulation running without issue.

## The Self-Driving Car Platform

The self driving car platform is implemented in the `AIController` class, which implements the IControl Interface and makes use of three different subsystems:

1. Sensing
2. Perception
3. Planning and Reacting

Each of these subsystems handles a third of the task of controlling the car autonomously. Firstly, the *Sensing* subsystem processes the world around the car, creating three different maps:

- A velocity change map
- A space map
- A colour map

These three outputs show the state of the world around the car in terms that can be processed and analysed. Secondly, the *Perception* subsystem analyses the three different maps to recognize objects of interest in the nearby surroundings. It is responsible for combining the three maps to identify objects based on known profiles and converting this information into `PerceptionResult` objects to be given to the final subsystem, the *Planning* subsystem.

The *Planning* subsystem takes the `PerceptionResult` objects generated by the *Perception* subsystem and, using this information, makes decisions about what action the car needs to take, if anything. The *Planning* subsystem makes direct calls back to the `AIController` to manipulate the car, accelerating, decelerating and turning.

This system can be summarised in the design diagram in Figure 2.

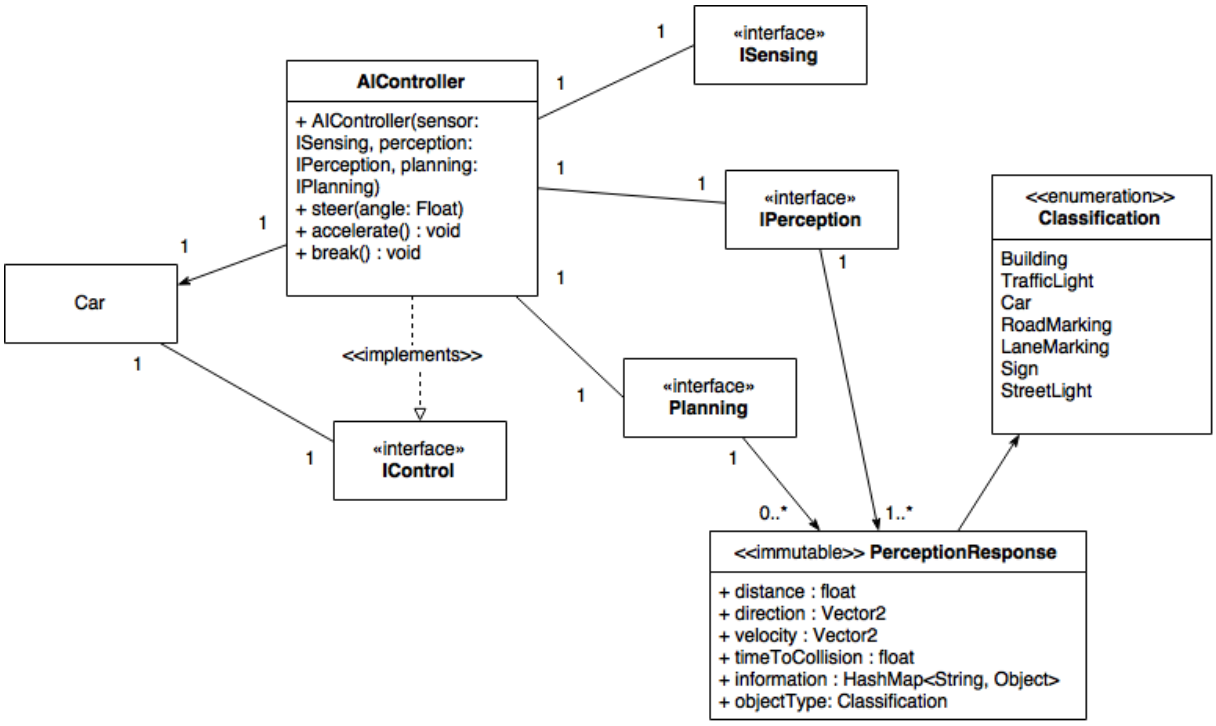


Figure 2: AI Controller Platform

### 1. The ISensing Interface

The sensing interface interacts directly with the world simulation, and will therefore need to interact with the classes from the system class diagram. It's interface is defined as follows:

```
// Update is responsible for updating the current state of the world, it should be
// called with a position and scan the world at that point in time. This then
// allows the users to retrieve the various maps from the other interface methods
+ update(Point2D.Double pos, float delta, int visibility): boolean

// Returns a two dimensional array containing the current relative velocity of any objects
// within the space. If there are no WorldObjects in the space, the velocity should be 0.
// If there is a world object in the space, the velocity should be the relative velocity
// of that object compared to the object we are tracking. As an example, an object that
// has the same velocity and is moving along side a car would have a relative velocity of
// 0, and the velocity of an object moving toward us will have a relative velocity of our
// velocity plus their velocity towards us.
+ getVelocityMap() : Vector2[][]

// Returns a two dimensional array representing whether the space is
// or isn't occupied by another collidable World Object.
+ getSpaceMap() : boolean[][]

// Returns a two dimensional array of the additive colour of everything in a given position.
// around the current user.
+ getColourMap() : Color[][]
```

Each of the map functions return a NxN array of the world around the current point, where N is defined as the visibility passed in to `update`. These functions build the map by stepping one unit in each direction to build the NxN grid and calculates the required value (velocity, space occupation, or colour) for that grid.

The classes that implement this are stateful, the results of retrieving the maps will depend on when the `update` method was last called, and what position it was called with.

### 2. The IPerception Interface

The perception interface takes the output of the sensing interface and analyses the maps to return an array of `PerceptionResponse` objects, as specified in the simulation class diagram.

```
// Analyses the given maps and returns an array of perception response objects. It
// analyses these maps using a fixed set of known objects.
+ analyseSurroundings(spaceMap boolean[][], colorMap Color[][],
                      velMap Vector2[][]) : PerceptionResponse[]
```

There is no fixed specification as to how this interface recognizes objects, the only limitation is that it **must** be able to recognize all objects in the Simulation world, including lane markings, traffic lights (and their state) and other cars.

The Perception interface is required to provide at least the following information in the information HashMap of the PerceptionResponse.

| Classification | Key | Value |
|---|---|---|
| TrafficLight | "State" | TrafficLight.State |

You are welcome to add additional information to the HashMap if you think it is beneficial to your design.

### 3. The IPlanning Interface

The planning interface is responsible for processing and acting upon the perception responses in such a way that it can achieve the self-driving car's aim: getting to a given destination without any incidents. To implement the planning interface the following methods must be provided:

```
// Plans a route to the given destination, or the closest possible point to
// that destination, in the world the car is operating. Returns true if a route can be found
// and false if there is no route that gets the user within 50 units of the position
+ planRoute(Point2D.Double destination): boolean

// Takes the perception responses and processes these perception responses to identify any
// neccesary changes to the current course of action. This must consider all road rules when
// determining actions, and can only provide one set of inputs to the car at a time
+ update(results PerceptionResponse[], float delta): void

// Returns the estimate of the time remaining until the car reaches its destination,
// given in seconds. If there is no destination, it returns 0.
+ eta(): float
```

The planning interface has direct access to the car and the AIController, and is therefore able to apply actions (steering, accelerating and braking) to the car and knows the car's current position at all times.

### Design and Implementation Advice

Some of the requirements provided for each of the subsystems can be implemented in drastically different ways, both from a design point of view and an algorithmic efficiency standpoint. You should remember here that your principle marks come from good software design. The runtime and space efficiency of your algorithms are much less important (in this subject) than the design decisions and rationale. Therefore,

you should ensure you optimize algorithms if and only if you have time to do so and have completed all other tasks.

You should remember that your implementations of the interfaces can be *stateful*. They will be a combination of classes, that can store instance variables and have constructors to pass in values from the outside of your subsystem. In a majority of cases, you will not be able to succeed unless you store previous state within your classes.

If, at any point, you have any questions about how things should operate or the interface specifications are not clear enough to you, it is your responsibility to seek clarification from your tutor. Please endeavour to do this earlier rather than later so that you are not held up in completing the project waiting behind a number of other students in the queue.

### Initial Testing of Your Implementation

Given your system will only implement one of the three subsystems, you will be required to test your subsystem in isolation during development. To assist with this, we will be releasing test utilities for each of the subsystems at the start of Week 10.

### The Task

Your task is to design and implement your assigned interface, in the context of the existing simulation. To this end, you will be completing three submissions. The first will specify your draft design for your subsystem which will be evaluated and given feedback in Week 10. The second will be your revised final design. Third will be your implementation of your assigned subsystem. You will be testing your implementation using the provided platform, and you will be attempting integration with the implementations of two other teams, reporting your results.

The exact task break down for these submissions follows.

### System Design

The City of Melbourne has provided you with a high level software design for the given simulation, and the specifications for a number of interfaces that will need to be implemented as part of this project. Your first task is to understand the provided design and specify your own design for your assigned interface.

This design should consider all relevant software design principles used in the subject thus far. You are required to provide formal software documentation in the form of Static and Behavioural Models.

### Static Models

You must provide a Design Class Diagram for your implementation of your assigned interface. This Design Diagram must include *all* classes required to implement your design including all instance attributes and all methods. You should consider the responsibilities you are assigning to your classes, as a general indication most interfaces should result in at-least 8 different classes, possibly more depending on how you choose to assign responsibilities.

### Behavioural Models

In order to fully specify your software, you must specify the behaviour along with the static components. To do this, you must produce 4 different sequence diagrams and a communication diagram showing how your system behaves. For your sequence diagrams, you must present the following 4 use-cases, depending on which interface you have been assigned to.

| Sensing Interface | Perception Interface | Planning Interface |
|---|---|---|
| Updating the scan | Combining the Maps | Calculating Routes |
| Generating the velocity map | Detecting Objects | Providing an ETA |
| Generating the colour map | Classifying detected objects | Prioritising `PerceptionResponse` |
| Generating the space map | Creating `PerceptionResponse` results | Handling Highest Priority Items |

Your sequence diagrams must be low level sequence diagrams that are consistent with your static models. Further to this, you must provide a communication diagram that shows all communication between your components within your subsystem.

### Design Rationale

Finally, you must provide a design rationale, which explains your choices made when designing your subsystem and, most critically, **why** you made those decisions. You may want to apply GRASP patterns, or any other techniques that you have learnt in the subject, to explain your reasoning. Keep your design rationale succinct, you must keep your entire rational to between 1000 and 2000 words

### Draft Design Submission

You are required, as part of the project, to provide a completed draft of your design as a first submission, one week before your Final Design Submission. After your Draft Design Submission, you will then be required to attend one or more workshops (as a team) in the following week to receive feedback on your design. This allows us to identify any possible major issues with your design, as well as provide feedback on particular aspects that may need to be remedied for your later submissions.

You will receive 1 mark for an on-time submission of a completed design draft, that is, a draft that includes all elements as specified in the submission checklist. Partial drafts will not receive the mark for draft submission, but will receive feedback during the week.

### Final Design Submission

You are required to revise your design according to feedback you receive and your own analysis of your draft. You will then submit a final design, a week after having submitted your draft design.

### Implementation Submission

Your final task is to provide a working implementation of your subsystem. You **must** provide this in an isolated package namespace so that it can be tested with other subsystems. This namespace should be of the form `{group number}.{interface}`, for example Group 91 working on Sensing would name their package `group91.sensing`. You must implement the interfaces provided within the simulation package. This implementation must be consistent with your submitted final design; if this is not possible due to a flaw in your final design, your implementation may vary from the design to correct this flaw and this inconsistency with your design must be clearly noted in comments in your source code.

You should ensure that your system provided is well commented and follows good Object Oriented principles.

### System Testing

As part of your implementation you must test your system with at-least 1 of each of the other subsystems that you haven't implemented. This type of integration testing will help highlight any issues with your implementation. As part of your task you must write a brief (200-300 words) report on your findings when testing your subsystem with other groups systems. You must include in this the group numbers of the group you tested with the system and the success of these tests; were your systems able to work together or did you discover issues in one or more of the subsystems?

> **Note**: this will require a working subsystem, so you will want to start on implementation earlier rather than later to ensure you have sufficient time to complete this task.

### Version Control

It is **strongly** recommended that you use *Git* for version control, for example, using the free private repositories available on bitbucket (https://bitbucket.org/). Using a version control system makes it much easier to work in a team environment on a complex project project. If you do use so, please provide us with access to the repository by adding Mat (on github: matblair, on bitbucket: mblair).

If you do use version control, please ensure you have set your repository to **private** so that other students in the subject cannot find and copy your work.

### Building and Running Your Program

Your program must be importable using the same instructions we have provided you to run the project. We will be testing your application using the desktop environment.

> **Note** Your program **must** run on the University lab computers. It is **your responsibility** to ensure you have tested in this environment before your submit your project.

### Submission Checklist

This checklist provides a comprehensive list of all items required for submission for the each part of this project. Please ensure you have reviewed your submissions for completeness against this list.

### Design (Draft and Final)

1. Ensure you have added your group number to all documents
2. Create a ZIP file including the following:
    a. 1 Design Class Diagram for Subsystem Design
    b. 4 Sequence Diagrams (outlining the functionality specified for your interface)
    c. 1 Communication Diagram
    d. 1 Design Rationale (1000-2000 words)

### Implementation

1. Ensure you have added your group number to all documents
2. Zip your package folder including all required src files.
3. Integration test report including:
    a. The groups you tested with and which subsystem they developed
    b. The results of that testing

### Marking Criteria

This project will account for 15 marks out of the total 100 available for this subject. These will be broken down as follows:

### Draft Design

The Draft Design Submission counts for 1 of the 15 marks available for this project.

### Final Design

The Part I Design Submission counts for 9 of the 15 marks available for this project, broken down as follows:

| Criterion | Mark |
|---|---|
| Class Diagram | 2 Marks |
| 4 Sequence Diagrams | 4 Marks |
| Communication Diagram | 1 Mark |
| Design Rationale | 2 Marks |

We also reserve the right to deduct marks for incorrect UML syntax and inconsistencies within your diagrams.

### Implementation

The Implementation Submission counts for 5 of the 15 marks available for this project, broken down as follows:

| Criterion | Mark |
|---|---|
| Interface Compliance | 1 Mark |
| Integration Testing Report | 1 Mark |
| Functional Correctness | 2 Marks |
| Code Quality | 1 Marks |

We also reserve the right to award or deduct marks for clever or poor implementations on a case by case basis outside of the prescribed marking scheme.

Further, we expect to see good variable names, well commented functions, inline comments for complicated code. We also expect good object oriented design principles and functional decomposition.

If we find any significant issues with code quality we may deduct further marks.

> **On Plagiarism**
>
> We take plagiarism very seriously in this subject. You are not permitted to submit the work of others under your own name. This is a **group** project. More information can be found here: (https://academichonesty.unimelb.edu.au/advice.html).

### Submission

All submissions must be made to the LMS using the provided links on the project page. Your submissions for each part must include all items as required in the submission checklist. Your document submissions (whether they are design documents or reports) **must be pdf documents**. Documents in any format other than PDF *will not* be considered during marking.

You must submit one **zip** file for each submission containing all required items. You must not use any compression format other than **zip** for your submission. Other archive formats will not be considered for marking.

Every item you submit **must** contain your LMS Group number for identification purposes.

If you have any questions at all about submission, please contact your tutor *before* the submission due date.

Only one member from your group should submit your project.

## Submission Date

- Draft Design is due at **11:59 p.m. on the 8th of May.**.
- Final Design is due at **11:59 p.m. on the 15th of May.**.
- Implementation is due at **11:59 p.m. on the 29th of May.**.

Any late submissions will incur a 1 mark penalty per day unless you have supporting documents. If you have any issues with submission, please email Mat at mathew.blair@unimelb.edu.au, before the submission date.