Design Rationale

Project Group 63 (Dennis Goyal, Rishab Garg & Thomas Fowler)

Throughout numerous iterations and various attempts, we eventually settled on a strategy-based design with potential state-based-systems within each strategy. There are various tasks which need to be performed to navigate and escape the maze, each which can be in a different state in its execution. Our design consisted of IDrivingStrategy's belonging and being acted upon by a single IDrivingStrategyActor (implemented by our CarController). The basic idea here is that our controller would pass execution onwards to its current IDrivingStrategy (with care taken to maintain a decoupled system as discussed later in this document) and this strategy would, depending on its state, query the world or update the car (through the CarController), or (if deemed necessary or an event occurs that is out of scope for this specific strategy) request that our actor transitions to another state.

We ended up with 6 basic strategies. One for traversing and exploring the world (which is the default strategy used). One for braking and slowing down. One for aligning to both a specified rotation and grid position. One for becoming 'unstuck' if our car becomes pegged up against a wall, and one for finding and waiting at a health trap if it is determined that we need to heal.

Our CarController subclass, obviously, is the controller in our submission. It is responsible for handling update events given by the simulation framework (and potentially an actual trial in the real world), maintaining state which is required throughout numerous strategies (such as a default IPathFinder and IPathCalculator) as well as responding to state transitions (where the state in this instance is the current strategy being used).

We chose to abstract away how we dealt with paths during our exploration of the map. Initially we stored computed paths as lists of coordinates, but eventually settled on the DrivingPath class. Internally this path just holds a LinkedList of coordinates, much like our original design, however this way we have the flexibility to represent a path any way we see fit. It was our original intention that each IPathFinder would have a custom IPath implementation (where IPath would be our abstraction over a path in the world) which would be created to best suit the nature of the path finding strategy in question (e.g. an IPath that stores a series of 'shapes' or one which stores a parametric function that would move through the space of the map). Eventually realising that we would only have one concrete implementation of DrivingPath (for any IPathFinder we considered), we dropped the inheritance but kept the factory pattern. Each IPathFinder still responsible for creating its DrivingPath. Incidentally, our DrivingPath class is also an example of protected variations, by not directly exposing its internal data structure (e.g. when querying for all coordinates in a path it returns a Collection and not the underlying LinkedList), it protects the internal list from being manipulated in an uncontrolled manner.

Our choice of abstraction was highly motivated by achieving a high level of cohesion. We separated out any element(s) we could into appropriate and well-defined modules which can then be injected into other modules as needed. A primary example of that is in our graph package, particularly with IPathCalculator and IPathFinder. Each of these modules has exactly one purpose. IPathFinder is responsible for finding and evaluating a path through the maze while IPathCalculator is responsible for working out the most efficient path. In general all path finding strategies will use a path calculator internally when evaluating the paths.

Indirection

Wanting to separate our different strategies for navigating the map, we had one problem: Coupling. Each and every strategy we designed for controlling the car would have to use our MyAIController as an interface for the Car. To decouple these, we decided to separate out the concerns of a CarController into two contracts: ISensor and IControls. ISensor being responsible to retrieving information about the Car's current view and our remembered information about the map, and IControls being responsible for the actual manipulation of the Car itself. The interfaces were designed in such a way that any CarController can implement them and not have to additionally override all of their specified methods. Using this technique, we were able to uncouple most code from our CarController to instead use either an ISensor, IControls or both. As previously mentioned, we have a strategy-based design where each IDrivingStrategy (owned by a IDrivingStrategyActor) has the ability to transition to a new state through the IDrivingStrategyActor owning them. This similar concept helped to reduce the coupling with our controller in our strategy system.

While not used in our final implementation, our original design for strategies involved copious amounts of inheritance and polymorphism. Our general idea was that many different strategies would have similar roles (at least in part) and as such, should be built in an inheritance tree to most efficiently define their actions. However, this causes several headaches, even though different strategies did have similarities amongst them, by relating them too closely there began to be serious scope infringements between them which causes conflicts when changing between states.

Additionally, we designed a utility interface (IMapIntelligence) which is fulfilled by our controller to hold the default instances of our Chart, IPathFinder, IPathCalculator (among others) and provides duplicate methods for those defined in the classes and interfaces it holds. This use of pure fabrication allows modules which need access to a path finding strategy and the chart (for example) to reuse previously created versions of these instead of individually requiring a unique instance. Additionally, the low coupling provided by this class allowed for a more efficient work flow when classes such as IPathFinder were being developed by one group member while another used the IMapIntelligence proxy to interface between them all.