SMD - Project C Design Rationale

Team (Group 17)	Hao Le (695540), Daniel Porteous (696965), David Stern (585870)
Description	Software Modelling & Design Project C, Semester 2, 2017
Deadline	Sunday 15th October, 11:59 PM

Introduction

This Design Rationale Report explains the thinking and design choices made by our team in the design of a software system designed for a single use-case: to automatically guide a car (*actor*) to navigate through a maze with traps to the exit of the maze (*goal*). We break this down by responsibility, examining the different possible assignments of responsibility and different options available at each point in our design, and why the end-decision was made. If a General Responsibility Assignment Software Pattern (GRASP) was applied, this too is documented.

Handling the Use-Case Scenario

The class being designed needs to receive system events of the use-case scenario; that is, updates in time, and the changing surroundings of the car that come with that. A natural approach to handling this kind of responsibility is the use of the Controller pattern, (a GRASP) wherein one overall system or root object class takes control of handling the system events of the use-case scenario.

The previous software implementation also uses this, in the form of the `AIController` that extends the abstract `CarController` class. However, this single controller class is a complicated, bloated controller; it performs many necessary tasks and responsibilities without delegating, meaning it has low cohesion and is unfocussed.

Thus, the solution (and chosen alternative) is to use a Controller class (`MyAIController`) that sufficiently delegates its responsibilities. This delegation effectively breaks up the system components into a modular system, in accordance with good Object Oriented Design principles. The main responsibilities identified were (1) Perception of the Environment, (2) Path Finding, and (3) Path Following (actual manoeuvring of the car; both steering and acceleration/braking).

This division of responsibilities also helps comprehensibility of the system; `MyAIController`'s `update` method now contains only the high-level logic of the system and shows how it makes its decisions.

Thus, `MyAIController` is responsible for the creation of `Sensor`, a single instance of `IPathFinder`, a single instance of `IPathFollower`, and a stack of instances of classes implementing `IPathFinder`. Its `update` method updates the sensor, updates the active PathFinder (popping a new PathFinder from the stack if necessary), feeds the sensor data to the PathFinder's `update` method, which returns coordinates that are then fed, along with the sensor data, into the PathFollower's `update` method.

Perception of the Environment

If `MyAIController` is to delegate its necessary tasks, an important one is the perception of the car's environment. `AIController` inherits from `CarController` methods that give access to information describing the car's environment and is therefore the existing information expert for the car. Following the Information Expert pattern is what gave rise to the bloated `AIController` class, and thus this pattern will not be followed. Instead, we have delegated the responsibility to the `Sensor` class, which retrieves all of the environmental information from the `CarController` and packages it in a `SensorData` object that is passed to the classes that require the information generated. This is an example of the Pure Fabrication GRASP; we are assigning a highly cohesive set of responsibilities to a class that is not in the problem domain (no actual 'sensor' exists) to support high cohesion, low coupling, and reuse.

Finding-Following 'Interface' Language

To maintain high cohesion, path finding and path following responsibilities were divided into two classes with these more focused sets of responsibilities. These classes are implemented as Java Interfaces, so as to maximise extensibility - more on that later.

Given this extensibility, the differing PathFinders and PathFollowers represent a point of instability or variation. Different PathFinders and PathFollowers should be able to work together. In order to do this, while dealing with the instability, we have defined an 'interface' language that succinctly communicates where the PathFinder wants the PathFollower to go. This is implemented via the `IPathFinder` interface's abstract `update()` method returning a coordinate, which `MyAIController` feeds through to the class implementing the `IPathFollower` via its abstract `update()` method. This coordinate communicates direction (via the difference in degree to where the car is currently facing) and desired acceleration (via how close or far away the coordinate is from the car). This is an example of a Protected Variation GRASP.

An example: How might the instances of `IPathFinder` communicate to those of `IPathFollower` that it needs to reverse slowly? If a car is facing north and is at coordinate (x=5, y=5), and assuming in this coordinate system (0, 0) represents the bottom left of the map, a coordinate of (5, 4) communicates to the IPathFollower to reverse (or turn around and move forward) slowly, whereas (7, 7) communicates to the IPathFollower to turn right while moving fast.

Path Finding & Path Following

PathFinding Responsibilities

PathFinder is responsible for taking the sensor data, and deciding on a path for the car to take. A PathFinder could be a class like `PathFinderExplore` that hugs the left wall, and keeps track of any sections of contiguous traps that it finds to its left, until it returns to the starting point before evaluating which trap section is best, and then adding `PathFinderEscape` classes that direct the PathFollower towards specific trap sections to the PathFinder Stack (see below) in reverse order (i.e. 'best' trap sections, like a 4x4 trap section of only Grass, would be placed on the Stack last).

Path Following Responsibilities

Classes implementing `IPathFollower` receive sensor data and a coordinate to follow, and using the MyAIController object they receive in their constructors have the responsibility to tell the car to accelerate forwards or backwards or brake (if at all), and which direction to turn (if any). A PathFollower class (PFC) achieves this responsibility with an `update` method that, based on the provided coordinate and sensor data, calls methods that control the car. As a result, it is a highly cohesive class, purely focused on following its path in whatever manner it chooses.

APCs can be implemented, for example, as a class that only ever moves at a speed of 1 (slow), and turns on the spot (`PathFollowerRotatingGrandma`), or as a class that moves as fast as possible, breaking only when instructed by the `IPathFinder` coordinate, turning well in advance to maintain maximum speed (`PathFollowerSmart`), and turning on the spot instead of reversing if stuck in a narrow corridor.

PathFinding & PathFollowing: Design Choices & Justifications

PathFinding and PathFollowing have both been separated into their own entities so as to focus responsibility and provide higher cohesion, and implemented as a Java interface to provide extensibility, so that if a new PathFinding algorithm or PathFollowing algorithm was to be used, it could easily be added as a new class that implements `IPathFinder` or `IPathFollower`. This is an example of the Polymorphism GRASP.

Justification: The Polymorphism GRASP reduces coupling (by not requiring any change to the system when a new pathfinder is to be created) and enhances extensibility. Implemented via an interface or abstract class, it requires the programmer to write only classes that conform to the aforementioned interface language. Thus it makes it easier for a developer to come along and know what they need to do in order to implement their own PathFinder or PathFollower, improving extensibility and allowing the aforementioned Protected Variation Pattern.

Alternatively, an abstract class could be used. A abstract class would make sense if there were any attributes or implemented methods common to all PathFinders or PathFollowers, however there are none, so an interface makes more sense.

Note: *Separation of responsibilities into these entities has been justified in 'Handling the Use-Case Scenario'.*

Traversal of Traps

`IPathFinder` has been broken down further for trap traversal by implementing an `TrapTraverse`

abstract class which implements the `IPathFinder` interface and can be implemented by trap traversal specific to each trap (e.g. fast acceleration before mud). This allows greater extensibility, such that complex groupings of traps can be handled by suitable trap traversal strategies. For example, if traversing a set of `Mud` traps and the car isn't travelling fast enough to clear the set of traps, the strategy could be to reverse back so that the car can get a run-up. This demonstrates both the Polymorphism GRASP, and high cohesion, so that more general classes implementing `IPathFinder` (like the aforementioned `PathFinderExplorer` need not deal with trap traversal-specific strategies.

PathFinder Stack

In order to track where the progress of our car through the simulation, we use a stack of PathFinders. Each PathFinder is responsible for a particular navigation job, such as an Explore PathFinder for finding possible exits from a "room" (a section bounded by traps), or a MudTraverse PathFinder for successfully navigating through a mud trapped section.

The stack represents the ordering logic for how the car transitions between stages of the navigation. The logic for how the car moves between them is represented by PathFinders creating new PathFinders and putting them onto the stack. For example, an Explore PathFinder might traverse a "room" and keep track of all exits. Once it returns to where it started, it orders the exits based on which looks least dangerous to cross, and pushes an Escape PathFinder for each of the possible escapes in inverse order of which is least dangerous (such that the least dangerous escape is on the top). This Escape PathFinder then pushes a Trap Traverse, which gets the car over the trap, which then pushes a new Explore PathFinder for the new room, and so forth. As for passing information between PathFinders, as an example, any information that the Escape PathFinder needs from the Explore PathFinder could be passed in via Explore's constructor.

This approach enables a high level of configurability in how the car transitions between navigation states, as opposed to the macro-level logic being hardcoded in some monolithic navigation class, as well as extensibility in regards to which strategies are used in the first place. We're aware that the stack is an artefact of our implementation, but we will briefly discuss the alternative approaches. Firstly, we considered a PathFinder manager class that would hold a set of PathFinder classes and know how to switch between them. This could be an implementation of the State pattern. We decided that this would be too hardcoded, and coupled to the details of particular PathFinders, making extensibility difficult. The other option was a nested PathFinder structure where, for example, an Explore might contain a set of Escapes, which themselves contain a set of TrapTraversals, and so on. This was clearly not ideal, a long dependency chain makes it difficult to swap new implementations in and out. As such we decided on the stack, which closely mirrors the standard recursive graph navigation method that you might employ in a non object-oriented environment.

Discarded Ideas

`Manoeuvre`: It was considered that we might create a manoeuvre class that maintains a state and facilitates complex manoeuvres of the car, such as three-point turns, however we came to realise

that our current control mechanism is extensible enough to support complex manoeuvres already. For example, a three-point turn can be implemented or executed by the class implementing `IPathFinder` placing the point behind the car, and the class implementing `IPathFollower` can decide on how to get to that point, changing its orientation if it so desires.

`Navigator`: We had implemented a `Navigator` class which managed the interaction between `IPathFollower` and `IPathFinder` classes, and contained the Stack and any additional navigation information shared between `IPathFollower` and `IPathFinder` classes. However, there seemed no reason as to why MyAIController could not perform the same tasks, while maintaining

Conclusion

This report has described the design choices made in designing a software system to automatically guide a car to navigate a maze with traps. Each responsibility of the system was considered, and assigned thoughtfully and appropriately, in line with the principles underpinning GRASP and Object Oriented design principles.