SWEN30006: Software Modelling & Design
Project Part C - Learning to Escape
Design Rationale - Group 37

Manindra Arora (827703)
Ninad Kavi (855506)
Ujashkumar Patel (848395)

Introduction

This design rationale report examines the existing framework provided to us and explains the thinking and design choices made by our team in the design of a software system designed for a single use-case. The aim is automatically guide a car (actor) to navigate through a maze with traps to the exit of the maze (goal). This has been broken down by responsibilities, by examining the delegation of responsibility and analysing 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, it has also been documented in the following report.

Use-Case Analysis

The class which we are required to design needs to receive system events of the use-case scenario, implement the interfaces provided as part of the framework and integrate with the existing framework. A natural approach to handling this kind of responsibility is the use of the Controller pattern, (a GRASP) wherein one overall system takes control of handling the system events of the required 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 quite complicated and does not delegate responsibilities well as it performs many of the necessary tasks and operations itself. Therefore it has low cohesion and is unfocused, which isn't characteristic of a good implementation.

Thus, our 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.) Exploring the map and identifying locations of keys
2.) Traversing the map to retrieve the keys in order
3.) Reaching the exit to complete the level

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.

MyAIController- High-Level Overview

If MyAIController is to delegate its necessary tasks, a critical responsibility 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, this responsibility has been delegated to the Sensor class, which which is used to obtain all the environmental and sensory data from CarController. Thefore this approach makes it easier for the classes which require access to the aforementioned sensory inforamtion. 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 reusability.

Explorer & Navigator: Design Choices

Explorer and Navigator have both been separated into their own entities so as to focus responsibility and provide higher cohesion, and have also been implemented as a Abstract Classes to provide extensibility, so that if a new Exploration algorithm or Navigation algorithm was to be incorporated, it could easily be added as a new class that implements Explorer or Navigator.

This is an example of the Polymorphism GRASP. The Polymorphism GRASP reduces coupling (by not requiring any change to the system when a new explorer or navigator strategy 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 Explorer or Navigator thus improving extensibility. Alternatively, an interface for Navigator and Explorer, could have been used instead of abstract class. An interface would be suitable in the event that each Navigator or Explorer required significantly different methods. However in our design, as they both require an update method, an abstract class made more sense

Explorer & Navigator: Delegation of Responsibilities

Explorer Responsibilities

Explorer is responsible for taking the sensor data, and choosing a suitable path for the car to traverse. A Explorer could be a class like SimpleWallFollower that always follows its left wall, or something more complex that uses Dijkstra's or A* path finding algorithms. These coordinates are then relayed onto the Navigator to be traversed and explore the remaining map.

Navigation Responsibilities

Classes implementing Navigator receive sensor data and a list of coordinate to follow generated by the Explorer class.Using the MyAIController object they receive in their constructors, Navigators have the responsibility to tell the car to accelerate forwards, accelerate backwards, apply brakes and which direction to turn and so on. A Navigator class 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.
Navigators can be implemented in many different ways to traverse the map. This can be done by a class that traverses the map slowly and steadily to account for all traps, or another class which makes its way to the exit as fast as possible to give a few examples, all based on instructions from the Explorer class

Proposed Strategies

In order to further improve our design we would have liked to add a Explorer Stack to be able to track the progress of our car through the simulation.
Each Explorer is responsible for a particular navigation job, such as an CircuitExit Explore for finding possible exits from a “room”, or a TrapTraverse PathFinder for successfully navigating through a trap section. This stack will represent the ordering of the changes in states of a car while exploring the map. The logic for how the car moves between them is represented by Explorers creating new Explorers and putting them onto the stack. For example, the aforementioned CircuitExit Explorer 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 instance of this class for each of the possible escapes on the stack, in an order such that the least dangerous escape is on the top.
Compared to the case where most of the logic is hardcoded in some monolithic navigation class, this approach enables a high level of configurability in how the car transitions between navigation states, and extensibility in regards to which strategies are used in the first place. We hence decided on incorporation such a stack, as it is quite similar to most the standard graph navigation methods, (which are usually recursive) that one might employ in a non object-oriented environment.

Conclusion

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