SWEN30006 Software Modelling and Design
Project 2: Parcel Pickup

School of Computing and Information Systems
University of Melbourne
Semester 2, 2019

Overview

Robotic Mailing Solutions Inc. (RMS) were so happy with your changes and design report for the Automail simulation system they decided to bring you onboard for a more challenging task. Wanting to break out of the bounds of delivering only within high rise buildings, *RMS* have decided to jump on the autonomous vehicle bandwagon and start a parcel delivery service. Right from the start they recognised that if they want to go global with their solution they will need to support different approaches that recognise local conditions.

RMS has provided you with a simple environment to allow you to demonstrate your design and development skills in controlling a parcel pickup vehicle with software-interfacing sensors and actuators. With your tailor-designed flexible integrated vehicle auto-drive and parcel pickup subsystem, this vehicle will navigate the world to locate and retrieve parcels, then take them to the *RMS* delivery depot.

The World

The area you find yourself in is constructed in the form of a simple grid of tiles consisting only of:

- Roads (some roads may lead to dead-ends)
- Walls
- Parcels
- Start
- Exit

Each environment only has one start and one exit, and the exit can only be used once a sufficient number of parcels have been collected.

The Vehicle

The car you find yourself in includes a range of sensors for detecting the car's speed and direction, and actuators for accelerating, braking and turning. It has only one speed (which can be applied in both forward and reverse) and can only turn towards one of the four compass points. *The car can only turn when it is already moving forwards or backwards.* The car also includes a short-range GPS that can detect/see four tiles in all directions (that's right, it can see through walls) – that is, it can detect the kinds of tiles in a 9x9 area centered on the car.

The car has a limited amount of health, and a limited amount of fuel. One unit of fuel is consumed for each tile the car moves onto, and the car is damaged (i.e., loses health) if it is driven into a wall. There is currently no way to regain health or fuel, but there may be in the future. If the car's health or fuel become zero, your software fails the test. If, however, your software takes the car from the start to the exit, picking up at least the number of required parcels on the way, your software passes the test.

The Task

Your task is to design, implement, integrate, and test a car auto-controller that is able to successfully traverse a map. It must be capable of safely:

1. exploring the map and locating the parcels,
2. retrieving the required number of parcels (or more), and
3. making its way to the exit.

It should complete these tasks before the car runs out of either health or fuel. At this stage *RMS* are **not** looking for complex or optimal algorithms for completing these tasks, but the solution you design should be well thought out and open for future extensions (e.g., to deal with hazards in the environment, or the make use of additional features in the delivery vehicle). You will not be assessed on how fast-traversing your algorithms are, just on the design in which they are embedded and on whether they work to get the car to the exit with an appropriate number of packages.

The Simulation Package

You have been provided with a starting point for building your solution. This package you will start with contains:

- a simulation of the world and the car,
- the car controller interface,
- a manual car controller that you can use to drive around a map,
- a simple car auto-controller that employs a wall-following technique, and
- sample maps.

The package also contains a file '`Driving.properties`', in which you can set the car controller, the run speed, the map, and other useful parameters. Note that when using some versions of Java, **when you run the program you may see a warning in red about an illegal reflective access operation; you can ignore this message**. You also **need to run the program in the context of the 'assets' folder**. That is, you need to set the working directory of the project to the 'assets' folder. In Eclipse you can do this by creating a run configuration, then setting the 'Working directory' from the 'Arguments' tab. The class '`DesktopLauncher`' in the package 'swen30006.driving.desktop' contains the main method for running the simulation.

If, at any point, you have any questions about how the simulation should operate or the interface specifications are not clear enough to you, it is your responsibility to seek clarification from your tutor, or via the LMS discussion board. Please endeavour to do this earlier rather than later so that you are not held up in completing the project in the final stages.

Project Deliverables

You have been provided with a design class diagram for the interface to the car controller and other simulation elements on which it depends. Your first task is to understand the provided interface and relevant aspects of the simulation behaviour.

You then need to specify your own design for a car auto-controller. As always, you should carefully consider the responsibilities you are assigning to your classes, and in particular, you should apply relevant principles/patterns covered in the subject thus far. You are required to provide formal software documentation in the form of static and dynamic models. Note that you are free to use UML frames to help manage the complexity of any of the design diagrams.

You may build on the existing autocontroller ("SimpleAutoController.java") or start afresh; either way, you will need to justify the end result (see 'design rationale' below).

Static Model

You must provide a design class diagram (DCD) for **your implementation** of a car auto-controller subsystem, **including the interface**. This design diagram must include *all* classes required to implement your design including all associations, instance attributes, and methods. It should *not* include elements of the simulation, other than your autocontroller and the interface.

You may use a tool to reverse engineer a design class model as a basis for your submission. However, your diagram model must contain all relevant associations and dependencies (few if any tools do this automatically) and be readable; a model that is reverse engineered and submitted unchanged is likely to reduce your mark.

Behavioural Model

In order to fully specify your software, you must specify the behaviour along with the static components. To do this, you must produce a communication diagram (CD) showing how the elements of your subsystem interact. That is, your communication diagram should show the **most important communications between components within your subsystem**. You **do not need to show message ordering** on this communication diagram.

Design Rationale

You must provide a design rationale, which details the choices made when designing your auto-controller and, most critically, **why** you made those decisions. In particular it must explain how your design supports the two variants (fuel and health) outlined above. You should refer to GRASP patterns, GoF patterns, or any other techniques that you have learnt in the subject, where relevant, to explain your reasoning. Keep your design rationale succinct; you must keep your entire rational to between 1000 and 2000 words. You may include diagrams in addition to the DCD and CD if it would help your explanations.

Implementation

You will submit a working implementation of your subsystem "MyAutoController" in the Java package "mycontroller". **You will not be submitting the entire system, just the "mycontroller" package**. The simulation and car controller interface must not be modified, as your mycontroller package will be inserted into the existing system for the purposes of testing.

Your implementation should be consistent with your submitted final design. We expect to see meaningful variable names, well-commented methods, and inline comments where appropriate. We also expect to see the application of good object oriented design principles and functional decomposition.

Peer Review

Each student will submit a confidential review of their own work, and the work of their team members. **This is a hurdle requirement** – failure to complete the review will result in a mark of 0 for the assessment. The peer review will be completed through an online form. More details will be made available through the LMS.

Testing Your Solution

We will be testing your application programmatically, so we need to be able to build and run your program without using an integrated development environment. The entry point must not change, and you must work to the provided interface. You must not change the names of properties in the provided property file or require the presence of additional properties. The auto-controller that will be assessed with be the class 'MyAutoController', in the 'mycontroller' package.

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 list of all items required for submission for this project. Please ensure you have reviewed your submission for completeness against this list. *Instructions for submitting will appear on the LMS, similar to Project 1.*

1. 1 Design Class Diagram (pdf or png)
2. 1 Communication Diagram (pdf or png)
3. 1 Design Rationale (pdf: 1000-2000 words)
4. A folder called "mycontroller" containing your car AutoController:
   (a) all the Java source files for your implementation of "MyAutoController"
   (b) only elements which are defined in the package "mycontroller"

Marking Criteria

This project will account for 15 marks out of the total 100 available for this subject. It will be assessed against the following rubric:

- H1+ [18.5-20] Solves all test maps. Good extended or extendable design/implementation with very consistent, clear, and helpful diagrams. Very clear design rationale in terms of patterns. Very few if any minor errors or omissions (no major ones).
- H1 [16-18] Solves all test maps. Good extended or extendable design/implementation with generally consistent, clear, and helpful diagrams. Clear design rationale in terms of patterns. Few if any minor errors or omissions (no major ones).
- H2(AB) [14-15.5] Solves most test maps. Extended or extendable design/implementation with generally consistent, clear, and helpful diagrams. Fairly clear design rationale in terms of patterns. Few errors or omissions.
- P-H3 [10-13.5] Solves some test maps. Reasonable design/implementation with generally consistent and clear diagrams. Fairly clear design rationale with some reference to patterns. Some errors or omissions.
- F+ [5-9.5] Plausible implementation and reasonable design with related diagrams. Plausible attempt at design rationale with some reference to patterns. Includes errors and/or omissions.
- F [0-4.5] No plausible implementation. Design diagrams and design rationale provided but not particularly plausible/coherent.

We also reserve the right to award or deduct marks for clever or particularly poor implementations on a case by case basis.

On Plagiarism: We take plagiarism and other forms of academic integrity very seriously in this subject. You are not permitted to submit the work of those not in your group.

Submission Date

This project is due at **11:59p.m. on Thursday October 24**. 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 Charlotte at charlotte.pierce@unimelb.edu.au, before the submission deadline.