Note to Radium: <TODO: To remove>

Bold for classes

Underline & Italic for concepts (GRASP, OOP, GoF)

Todo:

- src code:
  - Remove TODO comments
  - Clean up warnings
  - Remove unused classes
    - Testing subpackage
    - SpeedOptimisedRouteCompiler
  - Tweaks:
    - Use A* to explore as well
    - “g” cost should include time taken from start
    - When finding health tiles should consider how many lava tiles it has to pass as well as shortest distance

Introduction

The problem domain of this project consist of 2 parts (searching for keys and driving the car).

This is how we broke down the problem:

1. Searching: 3 types of **PathFinder**s
    a. Exploring Phase using a wall following algorithm until we find position of all keys <TODO Radium: may change to A* search>
    a. Finishing Phase using A* search to find a path to all the keys and then the finishing goal
    b. Recovering Phase to find closest health tile that has been seen using A* search when health hits a certain threshold
2. Driving: 5 types of **AutoPilot**s
    a. Maintain constant speed
    b. Turning 90 degrees
    c. Reversing for a certain distance (after colliding with a wall)
    d. Moving straight for a certain distance
    e. Recentring (re-adjust diverted path)

Components of our driving system (mycontroller)

1. **MyAIController** is a *Facade Controller* that monitors the state of the game similarly to a *State Machine* and delegates the work to PathFinder to find paths and gives them to Navigator to execute the instructions.
2. **AutoPilot**[1] subsystem's purpose is to abstract away the lower-level controls of the car and provides discrete, atomic operations to higher-level subsystem.
3. **PathFinder**s are responsible for finding the best routes depending on the stage of the game. The PathFinder system views the map as a discrete grid and outputs a path consisting of a list of coordinates to be on (e.g. (1,1) -> (1,2) -> (1,3) -> (2,3) ->...). We use the *Strategy Pattern* here by calling different PathFinder's depending on the state of the game
    a. **WallFollowingPathFinder** is used to explore the map for keys
    b. **FinisherPathFinder** & **HealthPathFinder** makes use of **AStarPathFinder** to find an optimal path to get all the keys and a health tile respectively
4. **RouteCompiler** component acts as a "translator" between PathFinder and AutoPilot subsystems by using the AutoPilot*Factory* to create AutoPilots for the given path. It is a *Creator*. The reason we have this, is because our AI/pathfinding algorithm outputs a list of coordinates, but our actuators (abstracted by AutoPilots) need more information to control the car (i.e. "Turn left at (1,1)" is more meaningful to AutoPilot than "Go from (1,1) to (0,1)"). Also, by grouping together coordinates on the path that are on the same line, we are able to achieve things like accelerating on a straight path.

[1] An analogy is the autopilot systems on airplanes: the autopilot systems are not completely autonomous, rather, it accepts input from human pilots ("go to altitude xxx metres"; "turn to yyy degrees") and decide the force to apply on the airplanes' thrust / flaps / wings etc. to reach the goal.

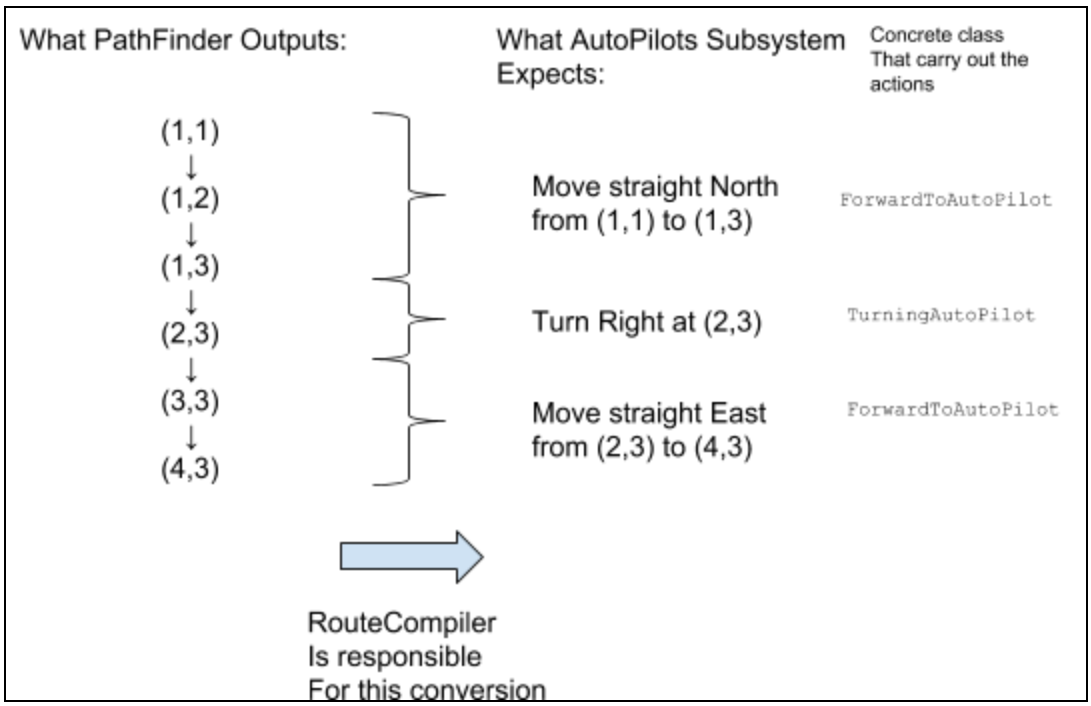

Figure 1: Illustration of the relationship amongst `PathFinder`, `RouteCompiler` *and* `AutoPilot`

5. **Navigator** component is a high-level “coordinator” of our driving system.
    a. Similar to a navigator on a ship, it asks its minions (i.e. AutoPilots) to do the heavy-lifting. It supervises AutoPilots, making sure that they can carry out their duties one-by-one and no AutoPilot is acting when it should not be activated.
    b. However, unlike a GPS "navigator", our Navigator does not do the path finding, that job is delegated to other subsystems
    c. The Navigator’s responsibilities are:
        i. To accept a path found by the PathFinder
        ii. To asks RouteCompiler to produce AutoPilots for our driving system based on the path
        iii. Keeps track of how far we have travelled on the calculated path (=> 100%)
        iv. When an unpredictable change occurs (*interrupts*), (e.g. car collided with wall, health below threshold, or current path navigation is done) it stops

the current navigation and re-routes (by calling relevant PathFinder's to find a new path).

6. **MapManager** component, is the "memory system" of our car driving systems. It is a *Singleton*. It remembers what the car has seen along the way and builds up information about the entire map in the process. Other components (PathFinder, and AutoPilot in particular) query MapManager to get information they need for decision making.
    a. We did not make MapManager a static class instead we used an Interface because we want the *flexibility & extensibility* of swapping out out different implementations in the future.
        i. For example, when unit testing our path finding algorithms, we can use a "mock" implementation of MapManager that loads the map from a text file (rather than getting it through exploring the interface). That way we can test our algorithms without the simulation system.
        ii. Also, by defining an interface explicitly, it is clearer to our team (other developers) what features a MapMananger should provide.

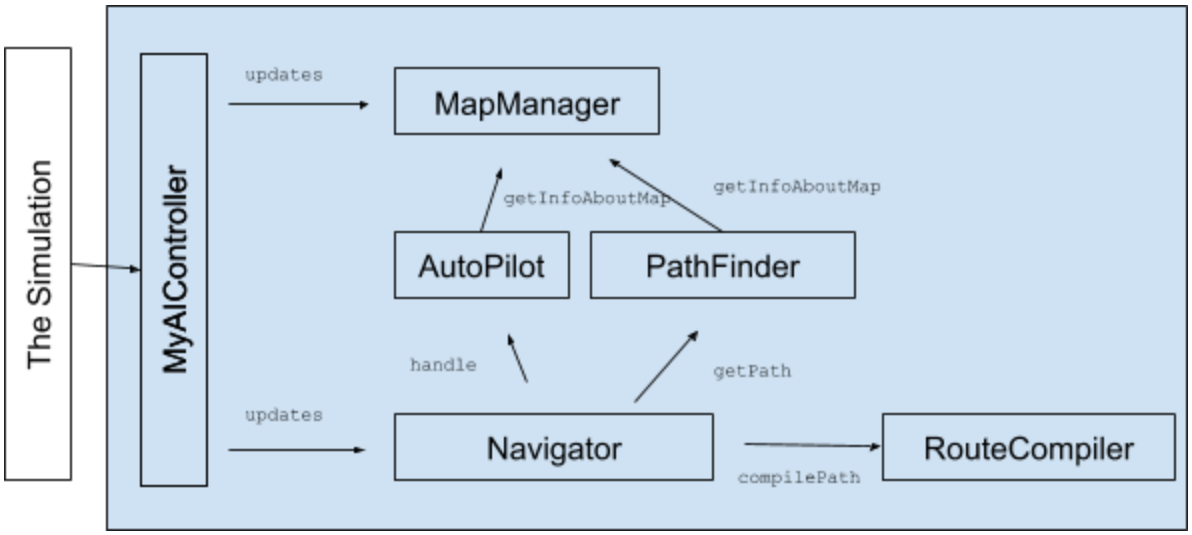

Figure 2: A conceptual overview of our subsystems (not UML). With all the responsibilities delegated, our MyAIController is basically just a mediator between our subsystems and the Simulation.

7. We used **ActuatorAction** (the low-level actions) and **SensorInfo** (the status of the car) as mediators to provide *indirection* between MyAIController, Navigator and AutoPilot. This way our Navigator and AutoPilot don't need to directly call methods like Controller.applyForwardAcceleration(). Hence, they don't have references to the Controller. This way we can achieve *low coupling and protected variation*. It also allows us to modify/filter the output of AutoPilot or combine multiple AutoPilots to a composite AutoPilot.
8. **Cell** class is our representation of the Tiles in the game, we wanted to design a cleaner and more concise way to access and check tile types:
    a. Without having our Cell class, to get position of the key from MapTile, we need to first check if it is a TrapTile and then cast it to TrapTile before calling getKey(). The MapTile.getType() method does not distinguish between HealthTrap and LavaTrap, so we need to use "instanceof" multiple times to determine its specific type. This way of accessing and checking is very hard to read and might lead to performance issues.
    b. An argument to this, is that we could have represented every data structure as plain strings and arrays for different type of tiles, but this would not allow for easy change if we needed to include more Tile Types, also it would be very cumbersome to find every instance of the String if we needed to change its name

Other details & design considerations

- **Util** class contains common reused methods and constants
- **Logger** class is for debugging, with varying degrees of verbose output
- We located **SortedList** & **Node** as nested subclasses in **AStarPathFinder**, as it was only used internally and helped reduce clutter.
- We did not use the Observer Pattern in which class? because it is not useful in this case. As there is constant updates being called. <TODO: Summarise>

Observer: is in the pathfinder. When should the autopilot be taking over control? Currently do it by constantly checking whether you has true. Who is the subscriber, observing the autopilot . it is the interface observes. The MyAIController is the user of autopilot and observing the autopilot. the event the autopilot can send out is that I should take over. When you send out that event, it will actually make the autopilot the one in charge and move the autopilot before. This is the observer (can be seen in the pictures) and it observes autopilot. How they observe the autopilot which is the interface? It observes everything inherited by autopilot. Basically in the autopilot interface, we may introduce different interfaces we may have register observer method and it will call that method. and this guy will know I have trouble and I need to notify but the problem is that this one is single term case. When you have one instance, we are going to have at most one observer. so there is no benefit to use the observer pattern.

Two benefits of observer:

1. when you have multiple subscribers, u do have to, you reduce some complexity
2. you reduce time u spent to do constantly checking state.

All the Classes in mycontroller package <probably remove this>

MyAIController.java

mapmanager:

- MapManager.java
    - DefaultMapManager.java

autopilot:

- AutoPilotFactory.java
- ActuatorAction.java
- SensorInfo.java
- AutoPilot.java
    - AutoPilotBase.java
        - ForwardToAutoPilot.java
        - EmergencyStopAutoPilot.java
        - MaintainSpeedAutoPilot.java
        - ReCentreAutoPilot.java
        - ReverseAutoPilot.java
        - TurningAutoPilot.java

pathfinder:

- PathFinder.java
- PathFinderBase.java
    - WallFollowingPathFinder.java
    - FinisherPathFinder.java
    - HealthPathFinder.java
    - AStarPathFinder.java
        - Node [private nested class]
        - SortedList [private nested class]
    - AStarHeuristic.java

navigator:

- Navigator.java
    - DefaultNavigator.java

routecompiler:

- RouteCompiler.java
    - RouteCompilerBase.java
        - DefaultRouteCompiler.java
    - SpeedOptimisedRouteCompiler.java

common:

- Logger.java
- Util.java
- Cell.java