ISearchAlgorithm, ISearchResult, Dijkstra, DijkstraResult, Node:

- Dijkstra, DijkstraResult, Node are responsible for implementing Dijkstra algorithm as well as record the result of it. In our design, the Dijkstra algorithm search for the shortest path based on a comparator, which is passed in as a parameter.
- This design is **easily modifiable and extensible**. For example, if there is more than one representation of the cost of a node, then by simply changing the comparator passed could get the desired shortest path. Furthermore, if a new way of searching is required, then this requirement could be satisfied by modifying the comparator only.
- Furthermore, the **strategy pattern** is adopted here to achieve an **extensible** design. For example, if we want to change the Dijkstra to another algorithm, this can be satisfied by creating a new class implement the ISearchAlgorithm.

MapRecorder:

- This class is responsible for recording information about the world, such as where is the lava and parcels. Furthermore, the recorded information is used for path finding later in **health or fuel conserve** manner.
- This class **increases the cohesion** of our subsystem because it is created based on the **pure fabrication** principle. The record of the tiles that the car has seen is *not a part of the domain model* and following the pure fabrication principle, creating a new class to take this responsibility would achieve a high cohesion. And this MapRecprder class takes the responsibility of recording the tiles has been seen by the car. Thus, this class increase the cohesion of our subsystem.

ITileAdapter:

- The getType method of different type of tiles is **not compatible**. Because the getType method of MapTile returns a TileType while the getType method for all the other tiles, such as WaterTile, return a String, this makes it difficult to record all explored tiles into one single HashMap for looking up later.
- Fortunately, this issue could be addressed by the **adapter** GoF design pattern. The purpose of the adapter is to provide a new common interface to several similar components but with different interfaces. Here, the getType method of MapTile and all the other tiles are similar since they are all responsible for returning the type of a tile, but as discussed above the return type are not the same. By following the adapter pattern, this ITileAdapter is created and defines getType method which returns TileType. As a result, with ITileAdapter, the not compatible return type of method is addressed without modifying any part of the original code.
- Furthermore, it is **easily modifiable**. In the future, if there is another new requirement requires String to be the return type, this could be achieved by modifying this adapter only without modifying any other part of the system.
- In addition, this interface also defines *isType* method which is used to check a tile is a specific TileType or not.

FinishAdapter, HealthAdapter, LavaAdapter, ParcelAdapter, RoadAdapter, StartAdapter, WallAdapter, WaterAdapter:

- All these 8 classes implement ITileAdapter, now the getType method of them all return TileType, which can be used in recording the tiles the car has seen.
- These 8 classes with the ITileAdapter together form an **adapter** pattern, and as defined in the lecture the adapter pattern follows the **pure fabrication**, **indirection** and **polymorphism** principles.
- These 8 classes are created based on the **pure fabrication** principle. Because all of these 8 classes are newly created to take the responsibility of returning the TileType of a tile, and none of them is in the problem domain. As a result, such classes make our subsystem **more cohesive and reusable**. For example, if another class want to use getType method and require TileType as the return type, it could use these classes directly.
-

TileAdapterFactory:

- There are 8 classes implements ITileAdapter, as a result, the logic of choosing which one to use is complex. To **expose this complex logic everywhere** is not a good and rational design since it is **not easily modifiable**. For example, if a new kind of tile called MudTile is added and the MudAdapter is created, then this adapter need to be included in the creation logic of ITileAdapter. However, the creation logic has been exposed everywhere, as a result, there are a lot of codes need to change which leads a **not easily modifiable** design.

- Our subsystem adapts the **(concrete) factory pattern** to address the issue above. This TileAdapterFactory is responsible for handling the creation logic of ITileAdapters, and it **hides the creation logic** instead of exposing the creation logic everywhere. This design is more preferable since it is more extensible. For example, if a new type of ITileAdpater is added, *only* the creation logic inside the TileAdapterFactory need to be modified.
- This class follows the **pure fabrication** principle because the creation logic of ITileAdapters is not a part of the problem domain and TileAdapterFactory is created to take this responsibility. Refer to the **pure fabrication** principle, this class increase the cohesion, and in reality, it is true. The TileAdapterfactory separates the responsibility of sophisticated creation into itself, which makes our subsystem **more cohesive**.
- However, this class may be called in many places, so many instances would be initialized. But it is not necessary to have many instances of TileAfapterFactory since they are all the same.
- The **singleton pattern** is applied to address this issue. The singleton pattern aims to provide global visibility of an instance to all the other objects. After making TileAdapterFactory into singleton, any object could access this factory directly without initializing a TileAdapterFactory locally.

IStrategy:

- There are several ways of choosing next Coordinate for the car to go depends on the situation. For example, if the car needs to be healed then it should move to the WaterTile or HealTie, or if the car sees a parcel then it should move towards the parcel. Thus, MyAutoController should be able to change the way of choosing the next Coordinate to move depends on the car's situation *at run time*.
- Refer to the **strategy pattern**, IStrategy interface is designed to enable MyAuto controller to **switch among different strategies** depending on the car's situation **at run time**. And this interface is responsible for finding the next Coordinate for the car to go depends on the current situation.
- Furthermore, such interface makes our subsystem **more extensible**, because if a new way of choosing next Coordinate to move is required, then create a new class implement this interface could satisfy this requirement without changing any other existing code.

ParcelPickupStrategy,HealStrategy, ExploreStrategy, ExitStrategy, RandomMoveStrategy:

- All of these five classes implement the IStrategy interface and each of them provides a *different* way of choosing next Coordinate for the car to move. In addition, these strategies *form the basis of health or rule conserves behaviour of the car*.
- *ParcelPickupStrategy* aims to find the next Coordinate to go which is on a path to a reachable ParcelTile.
- *HealStrategy* aims to find the next Coordinate to go which is on a path to a reachable WaterTilie or HealTile.
- *ExploreStrategy* is used when the car needs more information about the map. It aims to find the next Coordinate to go which is on a path to an outmost explored tile. An outmost explored tile refers to a tile which has already seen by the car and it is adjacent to at least one unexplored tile.
- *ExitStrategy* is used when enough parcels are collected and it aims to find the next Coordinate to go which is on a path to a reachable FinishTile.
- *RandomMoveStrategy* is used when the car has no destination to go and it aims to move the car to one of NORTH, SOUTH, EAST and WEST randomly.
- In addition, the next Coordinate that cost the least fuel is calculated using Dijkstra algorithm based on distance.

HealthConserveStrategy:

- This class is responsible for finding the next Coordinate for the car to go in ***health* conserve manner**. It implements IStrategy. Then, it stores some IStrategy in an ArrayList as well as manages them in **health conserve manner**.
- There are two comparators in this class, one is the HealComparator and another one is the HealthComparator. These two comparators ensure our IStrategy choosing the next Coordinate to move in **health conserve manner**.
- The HealComparator compare the cost of two nodes defined in the Dijkstra algorithm.
  - First, the node with **larger remaining health is better**
  - Then, if these two nodes have the same remaining health, the node with less fuel cost is better.
- The HealthComparator compares the cost of two nodes defined in the Dijkstra algorithm.
  - First, the node with **less health usage is better**
  - Then, if these two nodes have the same health usage, the node with less fuel cost is better.
- The getNextPath method in this class would return next Coordinate for the car to go in **health conserve manner**.
  - First, it determines whether it has collected enough parcel. If not it would determine is a path to pick up a parcel with the **Health**Comparator. If has collected enough parcel, it would determine whether there is a path to the FinishTile with the **Health**Comparator.

  - Then, if there is no way to pick up a parcel or exit, it would determine whether there is a path to an outmost explored tile with the **Health**Comparator.
  - Then, if there is no way to an outmost explored tile, it would determine whether there is a path to a WaterTile or HealTile with the **Heal**Comparator.
  - Otherwise, if there is still nowhere to go, it would ask the car to stay in the current position.
-

FuelConserveStrategy:

- This class is responsible for finding the next Coordinate for the car to go in **a *fuel* conserve manner**. Similar to the HealthConserveStrategy, this class also implements IStrategy. However, it stores some IStrategy in an ArrayList and manages them in **a fuel conserve manner**.
- There is a comparator in this class called FuelComparator, which ensures our IStrategy choosing the next Coordinate to move in a **fuel conserve manner**.
- The FuelComparator compare the cost of two nodes defined in the Dijkstra algorithm.
  - First, the node with **less fuel cost is better**
  - Then, if these two nodes have the same fuel cost, the node with the larger remaining health cost is better.
- The getNextPath method in this class would return next Coordinate for the car to go in **health conserve manner**.
  - The first step is almost the same as HealthConserveStrategy but change the HealthComparator with FuelComparator to achieve **a fuel conserve manner**.
  - Then, if there is no way to pick up a parcel or exit, it would determine whether there is a path to an outmost explored tile with the FuelComparator.
  - Otherwise, if there is still nowhere to go, it would ask the car to stay in the current position.

Composite Pattern:

- Refer to the **composite pattern,** Strategy, ParcelPickupStrategy, HealStrategy, ExploreStrategy, ExitStrategy, RandomMoveStrategy, HealthConserveStrategy and FuelConserveStrategy are created.
- This design makes the object using this class simpler since HealthConserveStrategy and FuelConserveStrategy take the responsibility of finding a Coordinate in **health or fuel conserve manner**, the object does not need to try about it.
- Furthermore, refer to the pure **fabrication principle**, this design makes our subsystem **more cohesive**. Because HealthConserveStrategy is created to be responsible for finding a Coordinate in a health conserve manner**.**

StrategyFactory:

- Refer to the **(concrete) factory** design pattern, this class is design to hide the creation logic of IStrategy. Currently, this factory handles the creation logic of the HealthConserveStrategy as well as the FuelConserveStrategy.
- Furthermore, for the same reason discussed in the TileAdapterFactory, the **singleton** design pattern is applied here to provide **global visibility** of StrategyFactory to all the other objects.
- For the same reason argued in the TileAdapterFactory, StrategyFactory follows the **pure fabrication** principle, and this **increases the cohesion** of our subsystem.

MyAutoController:

- In our subsystem, this class is design to controls the movement the car based on the next Coordinate return by the IStrategy, since this class is the **information expert** about moving the car.