Software and Modelling Design
Project 2 Report

Team W9-5

iTileAdapter, MapTileAdapter, TrapTileAdapter

In the provided design, *MapTile* class and *TrapTile* class have different interfaces for accessing tile types. Type of *MapTile* is obtained by calling *getType()* method, which returns a Type object that indicates the type of the tile. Whereas the type of *TrapTile* is obtained by calling *getTrap()* method, which returns a String representing the type of the tile. Also, types of tiles are accessed quite frequently. **Adapter** provides a stable interface. The adapter object converts the Tiles from the original interface to a stable interface. Hence, we think that it is better to unify the process of getting the tile type by applying the **adapter pattern**.

First, we created a unified class, *TileType*, which includes all types in the original *Type* class and all types represented by Strings in *TrapTile* class. Then we created an interface *iTileAdapter*. It has one method, *getType()*, for accessing tile type. All adapters then will implement this common interface so that the access point is unified. Every time an object wants to know the type of a tile, it can just simply ask the corresponding adapter, which is of type *iTileAdapter*, for type.

Without using the adapter pattern, the object needs to check if the tile is a map tile or a trap tile, and then call the correct method. There will be a lot of duplicate codes doing the same thing repeatedly. The system becomes very rigid and fragile: hard to maintain (a bug in a tile class results in changes to everywhere to ensure the system does not crash) and extend (in cases where there is a new type of tile added into the system, all places must be modified to use the new tile).

With the help of the adapter pattern, the system is more **extensible**. To use new tile types, new adapter classes can be created. While the existing adapter classes need not be modified, thus achieving **protected variations**. Moreover, as we assign responsibility to the adapter, the system has **higher reuse potential**. Direct coupling is also avoided because of indirection. **Polymorphism** is accomplished as different types of tiles are available through a single interface.

According to the principle of **information expert**, the tile itself is the expert. So, the responsibility of returning tile type should be assigned to the tile. This is done in the design provided. However, since the interfaces are not unified (as discussed above), simply following the information expert principle results in a bad design. To solve this, we turn to **pure fabrication** principle for help. We introduce the concept of adapter, which is not in the problem domain of this project, but helpful for reaching a high cohesion and low coupling design.

AdapterFactory, ControllerStrategyFactory, SearchStrategyFactory

Following the discussion of adapters, once they are ready, they can be instantiated and used. As mentioned above, tile types are accessed very frequently and widely (tile types are accessed in most of the supporting classes we created). That means that adapters are created everywhere. To solve this problem, we apply **factory pattern**. The factories we use are **pure fabrication** objects, we use them to handle the creation of adapters. As factory is not a concept in the problem domain, using factory increases the **representational gap**, but helpful for achieving a **low coupling**, **high cohesion** design.

AdapterFactory class creates the right tile adapter given the input tile. In *AdapterFactory* class, *getAdapter()* method returns a required adapter when being called. Now, all of “new” and “if-else” (for

deciding which adapter to create) will be replaced by a simple call to *getAdapter()*. In cases where the creation processes of adapters change, we only need to go to *AdapterFactory* class and modify accordingly.

Apart from the factory for adapters, we also have two factories for creating controller strategies and search strategies: *ControllerStrategyFactory* and *SearchStrategyFactory*. Controller and search strategies are used much less frequently, but creation processes are more complex, compared with adapters. Moreover, in this project, we are not required to implement algorithms that find optimal solutions. When better algorithms are used, we anticipate that the creation processes will become much more complex. Factories will then be much more useful as it hides the complexity of creation from other classes. Therefore, using **factory pattern** is a wise choice.

All three factories in our system are **singleton factories**, providing a single point of access. As discussed in the lecture, the **singleton pattern** is controversial because of its **global visibility** and should be used cautiously. We think that it is appropriate to apply singleton pattern here as one and only one instance of the factory object is reasonable in our system.

iControllerStrategy and Four Other Strategies

iControllerStrategy is the common interface in the system, implemented by all other strategies. The most important method it defines is *search()*. *ExitStrategy, ExploreStrategy, PickParcelStrategy,* and *HealingStrategy* are specific strategies for different scenarios.

We apply **strategy pattern** to deal with different types of strategies *MyAutoController* needs to perform its task. These strategies are related but have varying implementations. Each strategy is defined in a separate class and a common interface is provided for the calling objects. By doing so, generic algorithms are separated out from the detail implementations. That increases the **reusability** of code. In addition, these separate classes can be **easily extended** and **adapted**. Additionally, **strategy pattern** helps increase the **readability** of our code. Each strategy has a descriptive name, explaining the motivation of creation/intended purpose. As all strategies have the same type, *iControllerStrategy*, switching between different strategies can be done dynamically, thus accomplishing **polymorphism**.

If we do not apply strategy pattern, all the tasks are on *MyAutoController*. A class with a huge number of conditionals are expected, making it hard to understand and maintain. Changes in one place diffuse in the system and affect all other places, violating the principle of **protected variations**. The **open-closed principle** is violated as well. To change the implementation or extend for new functionality, the code has to be modified.

iSearchStrategy

This is the common interface for all search strategies. It defines a method called *search()*. All four controller strategies mentioned above make use of search strategies to find a solution that is considered desirable under their circumstances.

Without using the **strategy pattern**, we then need to write a helper method, which does the search, for each controller strategy. These methods are likely to be very similar (possibly created from copy and paste). This makes **system maintenance** extremely difficult. A bug in the implementation of the search

algorithm will affect all controller strategies, making it hard to spot the problem (perhaps, has to scan through all controller strategies) and fix the bug (fix it again and again for every place it appears). Besides, it decreases the extensibility of the system. Switching to a new search algorithm means the system is changed almost completely.

By using **strategy pattern** for search algorithms, each controller strategy class is more **cohesive**, not worrying about doing the search. The system achieves **reusability** as well. As search algorithms are all separate class, they can be used to do search in similar contexts in other systems. It is easy to find and fix bugs as there is only one place to go. New algorithms can be added into the system by creating new classes and having them implementing the common interface.

CompositeControllerStrategy, CompositeHealthControllerStrategy, CompositeFuelControllerStrategy

These three classes are the composite classes in our system, all of which implements the common strategy interface, *iControllerStrategy*. Using the same interface allows *MyAutoController* to use composite strategies the same way as for other strategies. As all composite classes have similar attributes, we define a common superclass, *CompositeControllerSrategy.* It maintains a hash table, which stores path cost for different types of tiles, a set of strategies and the type of search algorithm. All of these will be used by all other composite strategies. *CompositeHealthControllerStrategy* and *CompositeFuelControllerStrategy* are the two sub-classes inherited from *CompositeControllerStrategy*. As indicated by the name, we have one composite class for each mode of the car we are required to implement in this project.

Actually, in this project, composite pattern is not as helpful as it is in other cases. In our current implementation, health strategy and fuel strategy do not differ too much. These composite strategies are more for future improvements, for example, more strategies can be added to the composite class to accomplish more advanced functionalities in the future, or different composite strategies have exclusive subsets of strategies.

In terms of how our design support the two modes, each sub-class has a hash table, mapping from tile type to path cost of that tile. In health mode, as the health consumption is measured as the number of health taken, instead of the number of health consumed (taken by lava tile), our solution is to take health only when it is necessary (i.e. take or die). So, the path for health tiles (“water” and “health”) must be large, and Dijkstra will avoid these paths when returning search result. In fuel mode, we tell the car to always follow the shortest path possible, regardless of the types of tiles on the path. To allow Dijkstra to return the shortest path, we set all path cost to the same value (1 in our program). Each strategy has its own version of the path cost table and will pass it to search algorithm. This solution seems to be too simple, but it works fine this project. When the system becomes more complex, these two composite strategy classes may include two exclusive sets of strategies and perform totally differently. The true power of composite pattern will reveal then.

MyMap

This is the class in our system that handles all map related tasks. It contains all information about the map of the word, for example, places that the car has visited and the tile at a given position. It is responsible for initialising an empty map when the system starts and updating its own map using new information for every time step.

This class was not in the first version of our design. Initially, we put all the data and responsibilities into the *MyAutoController* class as this was in accordance with the **information expert principle**. MyAutoContorller knows everything about the world. It knows the initial configuration of the world. It knows what the car sees in every time step. Hence, by the **expert principle**, it should do all the jobs *MyMap* is currently doing.

However, by doing so, we reach a design that has **low cohesion**. Instead of focusing on controlling the car, *MyAutoController* has to pay attention to changes in the map as well. **Pure fabrication**, which is used in our adapter design, helps us again. We create this artificial class, and assign all map related tasks, which are highly cohesive responsibilities, to it. Now, *MyAutoController* can be concentrated on controlling the car. When it needs to do something related to the map, it will ask *MyMap* for help. Both classes are cohesive.