**DESIGN DECISIONS**

A brief summary of the various classes used in the software and their responsibilities:

|  |  |
| --- | --- |
| **Class** | **Responsibility** |
| **AbstractCityFactory** | An abstract factory for generating City instances. |
| **AbstractRouteFactory** | An abstract factory for generating Route instances. |
| **ActionConnectOperation** | The strategy which defines the algorithm for connecting cities on mouse press / drag / release actions in the drawable area. It extends ActionStrategy. |
| **ActionCreateOperation** | The strategy which defines the algorithm for creating cities on mouse press action in the drawable area. It extends ActionStrategy. |
| **ActionMoveOperation** | The strategy which defines the algorithm for moving cities on mouse press / drag / release actions in the drawable area. It extends ActionStrategy. |
| **ActionStrategy** | An abstract strategy for implementing the strategy to handle the mouse press / drag / release actions in the drawable area. |
| **CircleDecorator** | The concrete decorator in the decorator pattern, for decorating the city shape with a circle inside the central rectangle. It extends ShapeDecorator. |
| **City** | The data structure for city information. |
| **CityCenter** | The concrete component in the decorator pattern applied to decorating the city shape. It implements the ShapeInterface interface. |
| **CityCircleDecorationHandler** | The concrete handler in the Chain of Responsibility pattern, for applying circle decoration to a city. It extends CityDecorationHandler. |
| **CityDecorationHandler** | The abstract handler in the Chain of Responsibility pattern, for applying decorations to a city. |
| **CityFactory** | A concrete factory for generating City instances. It extends AbstractCityFactory. |
| **CityRepository** | The repository for City. |
| **CitySquareDecorationHandler** | The concrete handler in the Chain of Responsibility pattern, for applying square decorations to a city. It extends CityDecorationHandler. |
| **ConnectionClustering** | The strategy which defines the Clustering algorithm for connecting cities in the drawable area. It extends ActionStrategy. |
| **ConnectionContext** | The context class which controls the algorithm for connecting cities. |
| **ConnectionStrategy** | An abstract strategy for implementing the strategy to connect the cities in the drawable area. |
| **ConnectionTSPNearestNeighbour** | The strategy which defines the TSP Nearest Neighbour algorithm for connecting cities in the drawable area. It extends ActionStrategy. |
| **ConnectionTSPPro** | The strategy which defines the TSP Pro algorithm for connecting cities in the drawable area. It extends ActionStrategy. |
| **Container** | The container interface in the iterator pattern. |
| **EditCityDialog** | A dialog to let the user edit the city properties such as label, size, color and shape decorations. |
| **Iterator** | The iterator interface in the iterator pattern. |
| **Logger** | The logger class. |
| **MainFrame** | The main class for the application which contains the main() method. |
| **ObjectIterator** | A generic concrete iterator which implements Iterator. |
| **Route** | The data structure for route information between two cities. |
| **RouteFactory** | A concrete factory for generating Route instances. It extends AbstractRouteFactory. |
| **RouteRepository** | The repository for Route. |
| **ShapeDecorator** | The abstract decorator in the decorator pattern, for decorating the city shape with squares around the sides of the central rectangle. It implements ShapeInterface. |
| **ShapeInterface** | The component interface in the decorator pattern, for decorating the city shape |
| **SquareDecorator** | The concrete decorator in the decorator pattern, for decorating the city shape. It extends ShapeDecorator. |
| **TextAreaOutputStream** | It provides an output stream to write data to a JTextArea. It is used by the Logger for redirecting the output to the GUI. |
| **WorkSpacePanel** | The drawable area |

**Design Decision #1**: Implemented the **Factory** pattern for generating City and Route objects, to abstract the logic for creating the objects from the rest of the program. Also, both these factories implement the **Singleton** pattern so that the classes which need to access these factories need not define their own objects of these factories.

**Design Decision #2**: Implemented the **Strategy** pattern to dynamically change the action on mouse press/drag/release at runtime in response to user’s selection of Action i.e., Create, Move, and Connect, from the Action menu.

**Design Decision #3**: Implement Strategy Pattern for the various algorithms for connecting cities i.e., TSP – Nearest Neighbour, TSP -Pro, Clustering, and User Connect. The strategy pattern was used for giving the user the flexibility to switch between one algorithm to another during runtime.

**Design Decision #4**: Implement Observer Pattern for running the connection algorithm in background. This was done because every time a new city is added/loaded the connection algorithm (Observer) should automatically be notified by the classes which are handling the city additions (Observable). This notification from the Observable to the Observer triggers the algorithm to re-run and calculate the desired shortest route used the already in-place strategy.

**Design Decision #5**: Implemented the **Decorator** pattern for modifying the city shape. Here, we chose not to decorate the entire city class as that would involve extensive modification of the codebase. Instead, we used the decorator solely for modifying the shape of the city which by default, is implemented in the CityCenter class. Hence, we try to apply the open closed principle by restricting the number of changes made to preexisting code but adding functionality through additional classes.

**Design Decision #6**: Implemented the **Chain of Responsibility** pattern for applying the decorations to city shape through a chain of handlers which inspects some conditions, applies the decoration if the conditions are met and passes on the responsibility for decoration to the next handler.

**Design Decision #7**: Implemented the **Singleton** pattern for city and route repository so that the classes which need to access these repositories need not define their own objects of these repositories. Also, it ensures that there is only one single repository for each City and Route in the entire program.

**Design Decision #8**: Implemented the **Iterator** pattern for city and route list contained within their respective repositories. The city and route repositories are both containers which have the iterator pattern implemented with them. We are able to access the list of cities and routes through these iterators as we have shown in the paintComponent() function in the WorkSpacePanel.

**Design Decision #9**: Implemented the **Singleton** Pattern for logger so that the classes which need to log data need not define their own objects of logger. Also, since we are logging data to a common place, it makes sense to just have a single logger for the entire program which should be shared by all the classes.

**Design Decision #10**: Implemented the **Observer** pattern for drawing the cities and routes on the drawable area. The WorkSpacePanel class which is responsible for drawing the cities and routes is an observer of both CityRepository and RouteRepository, each of which extend Observable. Hence when there are changes in either of the repository, WorkSpacePanel is notified about it and it can take the necessary actions to update the GUI.