City University of Hong Kong

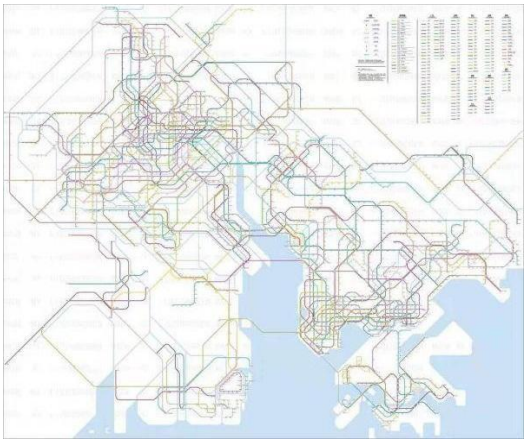
Department of Computer Science

CS3343 Software Engineering Practice

2021/22 Semester A

Hong Kong-Shenzhen Metro Route Planning System in Greater Bay Area

Analysis and Design Report



Project Group 6

HUANG Yuqin (56195499)

LIU Wei (56198489)

WANG Zhixuan (56198557, APM)

XU Jiakai (56197616)

XU Rui (56200188, PM)

ZHANG Xun (56198705)

TABLE OF CONTENTS

1 INTRODUCTION

Hong Kong-Shenzhen Metro Route Planning System is a system developed for civilians in Hong Kong and Shenzhen to enable route planning across the border.

In the past, if civilians wanted to travel across the cities, they should refer to the two maps respectively and it may bring some trouble for finding an appropriate route. Especially currently, the influence of the pandemic has made the administration introduce traffic control measures, which remarkably affects the commuting between two cities.

Thus, we expected to develop a new program integrating the two maps together, to bring an unified operation interface. Users can choose the start and end stations in two cities relatively, then the system will directly provide an appropriate route between two cities, which will make it more convenient for citizens to plan the route and travel, and therefore improve the commuting efficiency, which in turn promotes the coordinated development of the Greater Bay Area.

2 DESIGN CONSTRAINTS

2.1 Commercial Constraints

The time limit of the project is 13 weeks from the launch of the program on September 3. The time is relatively tight and some extensive functions cannot be done, so we can only simplify the design of the program.

Also our manpower is limited. There are only 6 members in our team, so it will require our team members to put more effort into their own responsible field. Members need to learn and be familiar with the target and make efforts to achieve it.

2.2 Compliance Constraints

Our program will integrate the metro network of Hong Kong and Shenzhen. However, the regulations on administration of two cities are different. So we should follow that situation, and consider how to retain the unity of two network systems and at the same time keep the independence and reduce the interference to some extent.

Also, we should ensure the formality and accuracy of the data source. We need to do a thorough early-stage investigation and find the data from the official data providers.

2.3 Functional Constraints

Our program should be equipped with following key functions according to the requirement:

- Users can choose the destination and departure stations (either in the same city or across the cities) and acquire the ideal path from the system.
- Administrators can have a clear partition of responsible stations and rail lines between two stations.

- Maintenance personnels can add or remove the stations and lines according to the top management of the corporation.
- The system should have different languages for different groups of potential users

2.4 Non-functional Constraints

On the premise of satisfying the above functions, we expect our program to optimize other characteristics:

- Fast to compute the route
- Easy UI actions
- Simple maintenance operations

2.5 Stylistic Constraints

Because of the limitation of the language (Java) and the lack of skills and experience of the members, we only expect to successfully reflect the relations between stations and lines on the UI, and we do not require other more stylistic designs.

2.6 Systems Constraints

Due to the time limit and user requirements, the system will only be executable on the Windows/Mac system. We do not provide the mobile-end planning system.

2.7 Skill Level Constraints

This is our first time to process such a general project, and we lack related experience.

For the use of the IDE and version control tools, we need to adapt them for some time. Also none of us had processed UI programming before, and none of us know about SWING, a GUI toolkit designed for Java.

Also we did not know about the knowledge of project management, and a good management will lead to smoother progress of future works. Therefore, we still need to do that to gain experience.

3 USE CASE DIAGRAM AND SPECIFICATIONS

3.1 Use Case Diagram

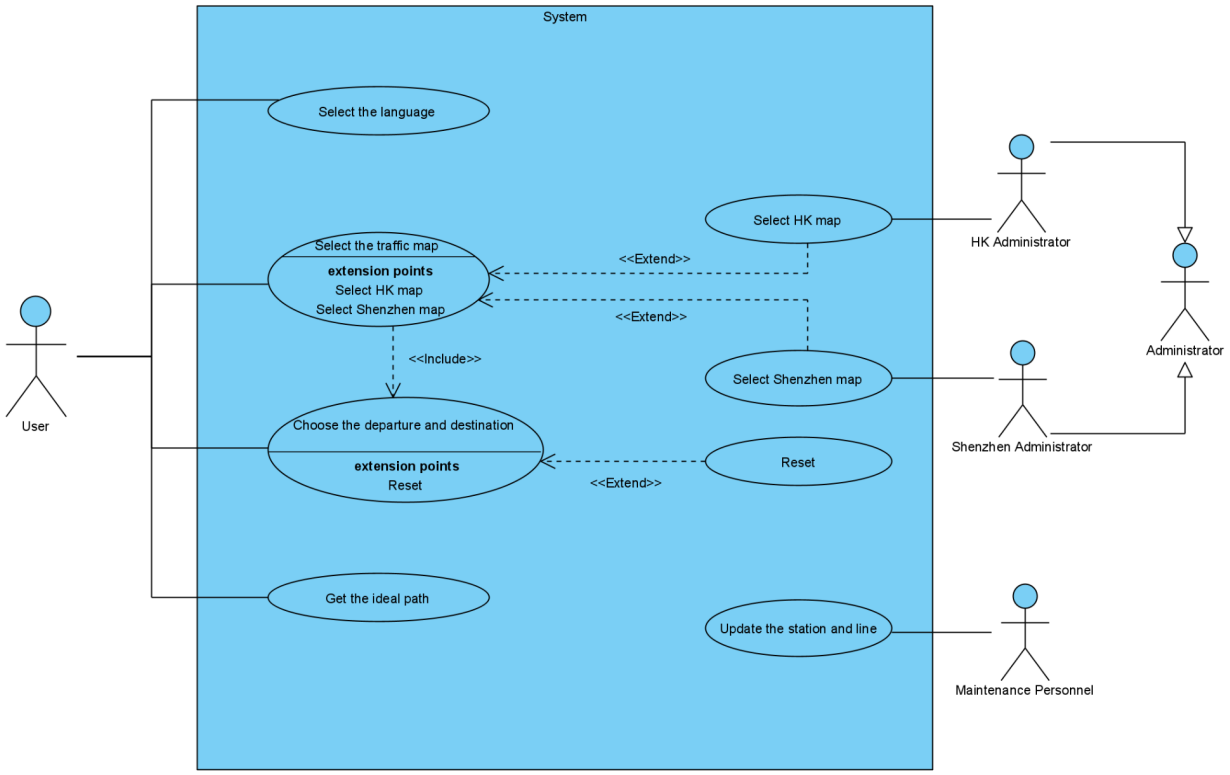


Figure 3.1 Use Case Diagram of Login Module

3.2 Use Case Specification

3.2.1 Select the Language

Use Case Name:	Select the language
Actor(s):	User
Description:	This use case describes the process of a User launching the program

Typical Course of Events:	***Actor Action***	***System Response***
	Step 1: This use case is initiated when the User selects this function.	
		Step 2: The system activates the interface for the user to choose their preferred language.
	Step 3: User choose the language by clicking the button	
		Step 4: The entire program will display with corresponding language.
Alternate Courses:	*Nil*	
Precondition:	*Users start the program successfully and the system functions are normal.*	
Postcondition:	*The setting is sent to the system and the user interface will show corresponding language.*	

3.2.2 Select the traffic map

Use Case Name:	*Select the traffic map*
Actor(s):	*User, Administration (HK and Shenzhen)*
Description:	*This use case describes the process of User choosing the metro map and the Administration arranging the jurisdiction area.*

Typical Course of Events:	***Actor Action***	***System Response***
	Step 1: This use case is initiated when the User selects this function.	
		Step 2: The system activates the interface for the user to choose a preferred metro map.
	Step 3: User choose the map by clicking the button	
		Step 4a: The system will call Hong Kong MTR administration when the user selects the HK map. *Step 4b: The system will call Shenzhen administration when the user selects the HK map.*
	Step 3: HK administration and Shenzhen administration get the latest information of stations and lines and send them to the system.	
		Step 5: The system will display the UI with the data.
Alternate Courses:	*Step 4a: If User selects the HK map, the HK admin will prepare the station data and be ready to calculate the shortest path with certain criteria.* *Step 4b: If User selects the Shenzhen map, the Shenzhen admin will prepare the station data and be ready to calculate the shortest path with certain criteria.*	
Precondition:	*The system functions are normal.*	
Postcondition:	*The user interface will show selected maps.*	

3.2.3 Choose the departure and destination

Use Case Name:	*Choose the departure and destination*	
Actor(s):	*User*	
Description:	*This use case describes the process of User choosing the departure station and arrival station.*	
Typical Course of Events:	***Actor Action*** *Step 1: This use case is initiated when the User selects this function.* *Step 3: Users choose the 'From Station' and 'To station' by clicking the button with the station's name.*	***System Response*** *Step 2: The system activates the interface for users to click the stations.* *Step 4: The system will send the data to the backend database.*
Alternate Courses:	*Step 3a: If User encounters any problem, they can reset the choice by clicking the reset button.*	
Precondition:	*The system functions are normal.*	
Postcondition:	*The shortest path generating backend system will get the core data.*	

3.2.4 Get the ideal path

Use Case Name:	*Get the ideal path*	
Actor(s):	*User*	
Description:	*This use case describes the process of User getting the shortest path with our system.*	
Typical Course of Events:	***Actor Action*** *Step 1: This use case is initiated when the User selects this function.* *Step 3: User gets the result.*	***System Response*** *Step 2: The system calls the backend system to calculate the shortest path and then display the result by highlighting the path.*
Alternate Courses:	*Nil*	
Precondition:	*The system functions are normal, and the departure and destination are valid.*	
Postcondition:	*The program runs successfully and the user will get the ideal path.*	

3.2.5 Update the station and line

Use Case Name:	*Update the station and line*
Actor(s):	*Maintenance Personnel*
Description:	*This use case describes the process of how to update the map.*

Typical Course of Events:	***Actor Action***	***System Response***
	Step 1: This use case is initiated when there is any update on the metro stations and lines.	
		Step 2: The system informs the Maintenance Personnel to add and delete the stations and lines.
	Step 3: The Maintenance Personnel updates the information by editing the related data file.	
		Step 4: The system gets the latest information and will display it the next time program is started
Alternate Courses:	*Nil*	
Precondition:	*The system functions are normal and the data information is correct.*	
Postcondition:	*The program runs with the latest version.*	

4 CLASS DIAGRAM DESIGN

4.1 Overall Class Diagram

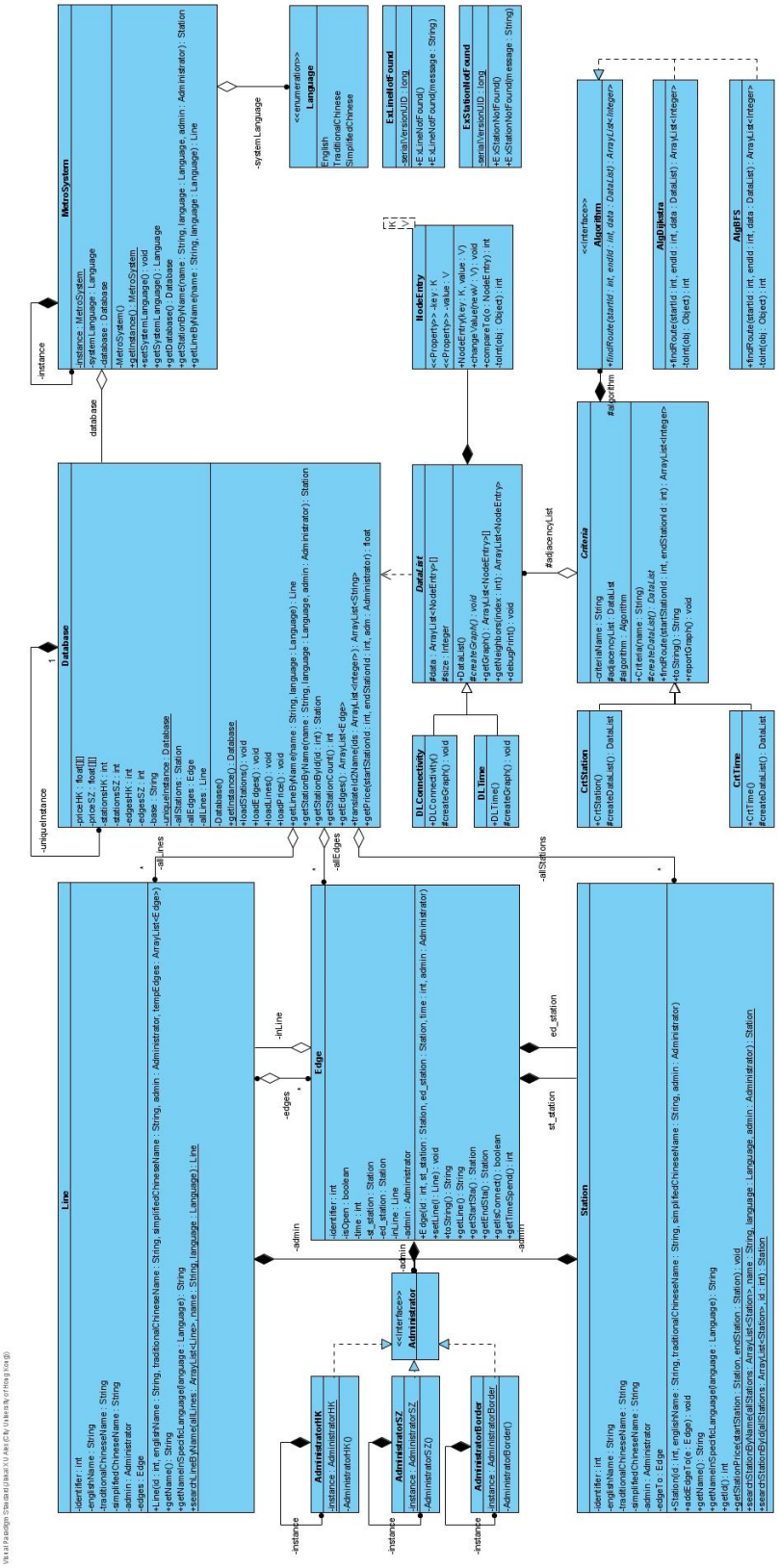


Figure 4.1 Class Diagram of metroSystem Package

4.2 Design Principles

4.2.1 Open-Closed Principle (OCP)

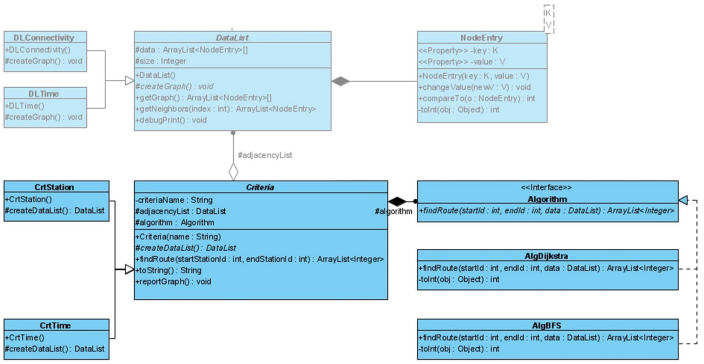


Figure 4.2.1 Detailed Class Diagram of Open-Closed Principle

Open-Closed Principle (OCP) states that classes should be open for extension but closed for modification, which is usually achieved by inheritance so that new subclasses could be easily extended [1, 2]. Compliance with OCP makes it easy to extend classes and incorporate new behavior without modifying existing code. Such a design is resilient to change and can accept new features to meet changing needs [2].

In our system design, we fully implemented the requirements of OCP, such as the "criteria" and "algorithm" in the above diagram is a typical example.

Our "criteria" are based on the "algorithm" superclass. We only need to know that "findRoute()" can calculate the correct route without knowing its internal working flow or specific codes of the "algorithm". When we develop a more suitable algorithm in the future, we just need to make this new algorithm class implement the "algorithm" interface, and then we can directly call it in the "criteria". Similarly, if a new criterion comes along, we can choose or develop an algorithm that will do the job. All of this is done without knowing the details of the existing code, or even making any changes, simply inheriting the corresponding superclass and calling it when appropriate.

4.2.2 Liskov Substitution Principle (LSP)

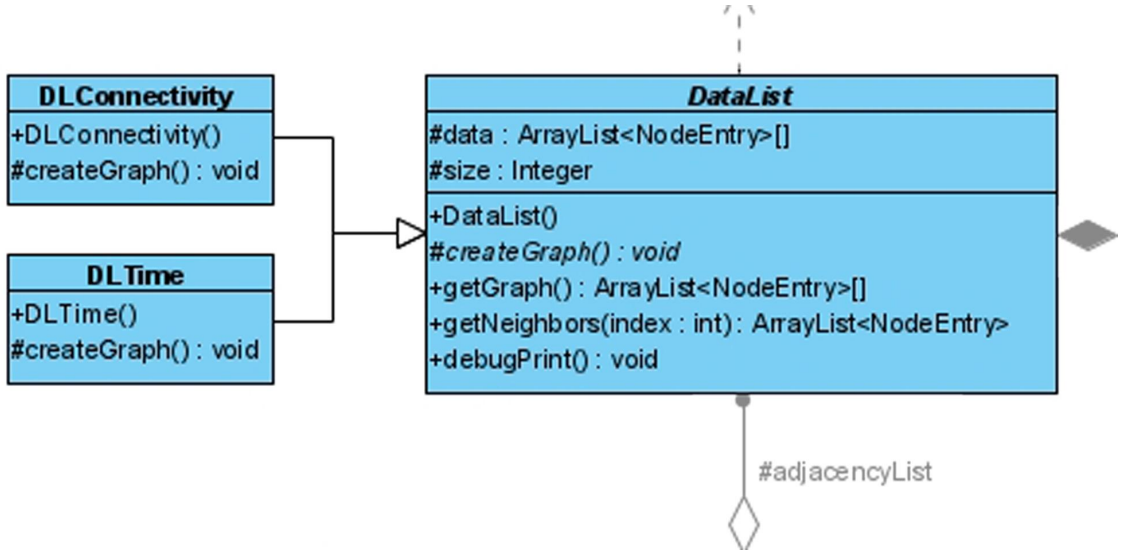


Figure 4.2.2 Detailed Class Diagram of Liskov Substitution Principle

Liskov Substitution Principle (LSP) ensures that newly derived subclasses only extend the base-class without changing their behavior. Subclasses should not inherit features that do not exist in the actual context [2]. In concrete, in our example, a subclass can implement an abstract method of the parent class, but cannot override a non-abstract method of the parent class.

We use the LSP principle to help us design the parent and child classes in the inheritance relationship. For example, concrete DataList methods only inherit their required methods without overwriting any non-abstract methods already declared in the superclass, i.e., without changing the behavior of the superclass, just extending the functionality of the superclass appropriately. So, if necessary, as required by the LSP principle, these subclasses can always replace the parent methods in our program without any impact.

4.2.3 Dependency Inversion Principle (DIP)

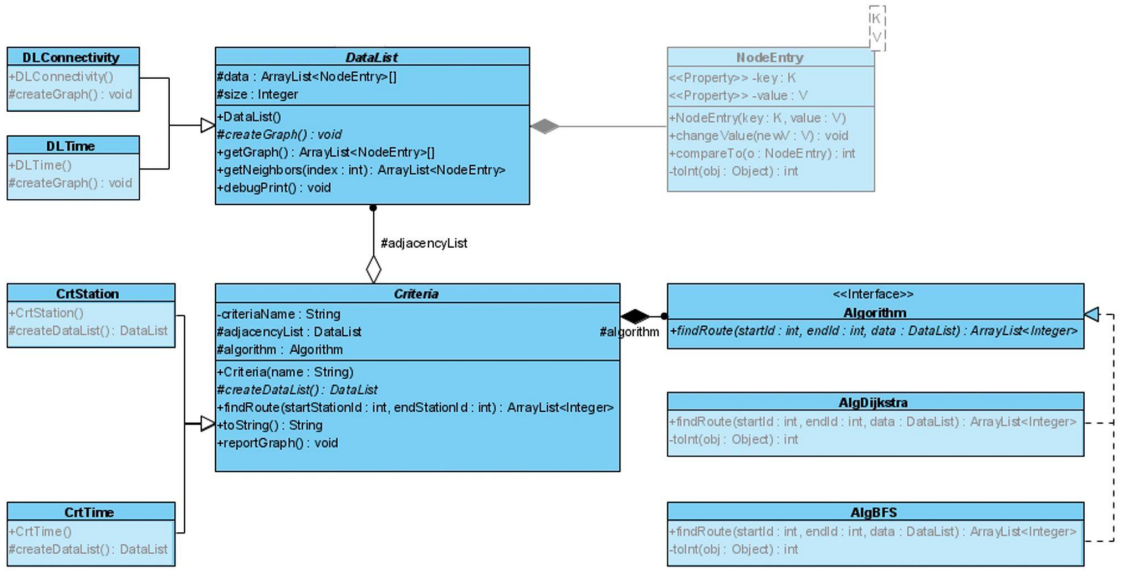


Figure 4.2.3 Detailed Class Diagram of Dependency Inversion Principle

The dependency Inversion Principle (DIP) states that high-level modules should not depend on Low-level modules. Both of them should depend on abstractions. DIP using abstract/virtual class or interface is a way to achieve OCP mentioned above [2, 4].

We keep this principle in mind when designing our projects, such as the "criteria - data list - algorithm" module at the back end, which is a traditional practice.

In this module, all programming activities are directed towards the "interface" rather than the specific underlying "implementation". That is, the high-level component "algorithm" only uses an abstract "data list" and does not care about the implementation of the concrete underlying "data list". Similarly, in the Factory method pattern, the "criteria" class requires only one variable that belongs to the high-level component, "data list", while the specific instantiation of one of the many compliant data lists is determined at run time depending on the particular situation.

Following this DIP design principle allows us to greatly reduce the dependency of our programs, making our software more flexible to adapt to future changes and challenges.

4.2.4 Law of Demeter (LoD)

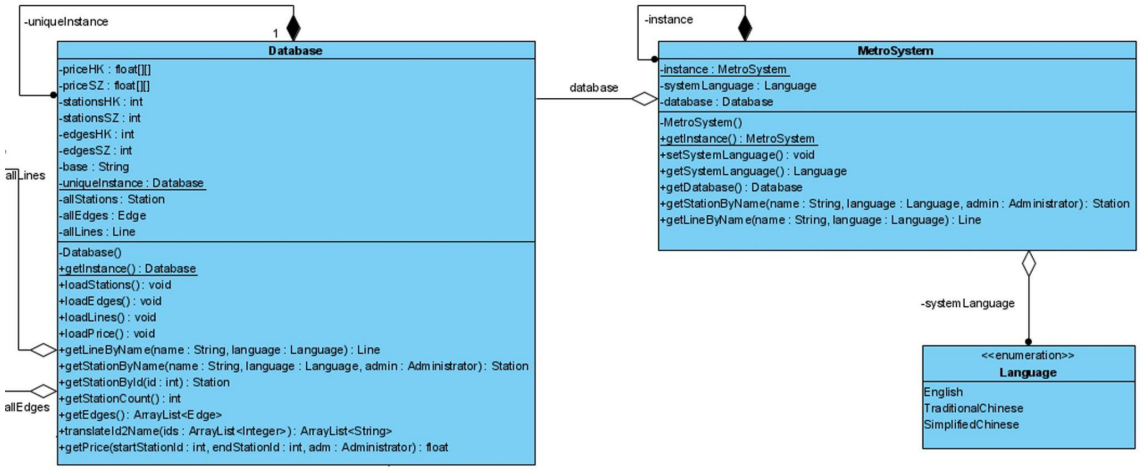


Figure 4.2.3 Detailed Class Diagram of Law of Demeter

The Law of Demeter is a design guideline for developing software applications. This principle states that an object should never know the internal details of other objects. [7]

In our project, we try to make the objects independent of each other as much as possible. The MetroSystem class only invokes itself and direct component objects. Its behavior of a specific method, such as getStationByName () is completed by itself, rather than pointing to another object Database to implement.

The LoD principle can reduce the dependency between classes and coupling, which is defined as the degree of interdependence that exists between software modules and how closely such modules are connected to each other.[6] Therefore, our software can be easily maintained, tested, and updated.

4.3 Design Patterns

4.3.1 Factory Method Pattern

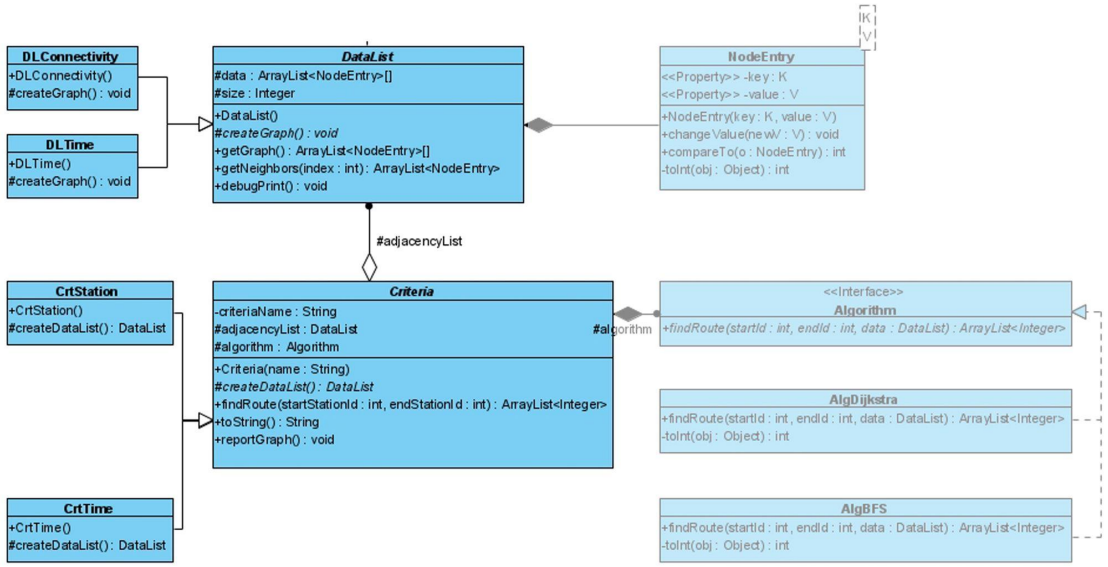


Figure 4.3.1 Example of Factory Method Pattern - the DataList related classes

The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses. [3]

We use the factory method pattern here to encapsulate the instantiation of a specific datalist type. As shown above, the abstract DataList class provides a factory method interface called "createGraph()" to create concrete objects. Any other methods implemented in this abstract class might use the concrete "data list" created by the factory method function, but only its subclasses, such as "DLConnectivity" and "DLTime", actually implement the factory method and create the concrete product, different types of the data list.

Using this pattern lets us write the creator class and all the other functionality associated with it without knowing which product is actually being created. In this way, the degree of dependence between different classes is significantly reduced, and the OCP principle is fulfilled to ensure the system's cohesion. When in actual use, which subclass of our system is selected will naturally determine what data list is actually created and applied.

4.3.2 Strategy Pattern

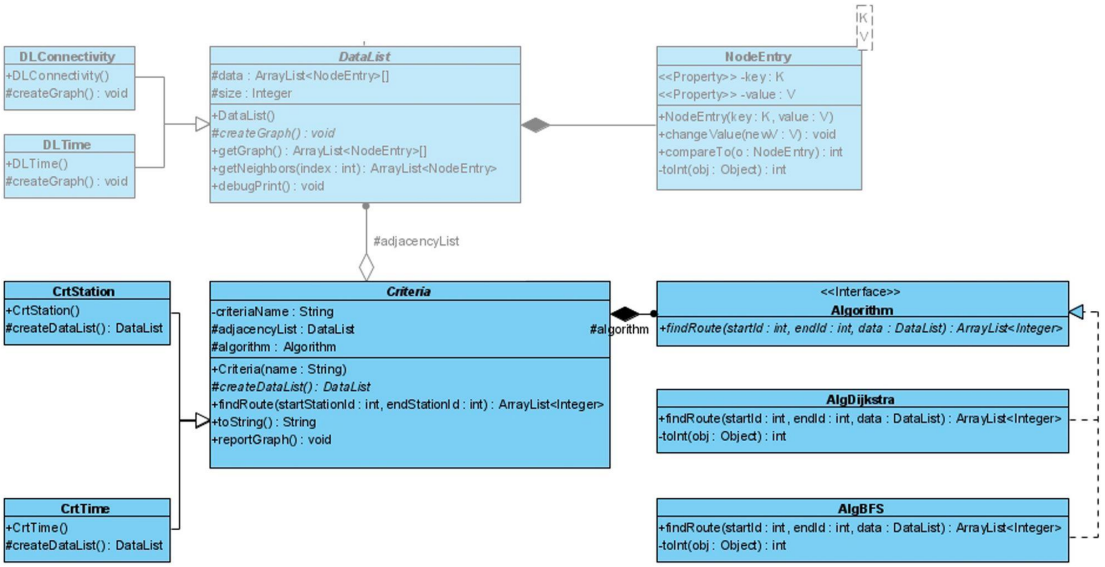


Figure 4.2.3 Example of Strategy Pattern - the Algorithm Class and its States

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from the clients that use it. [3]

Strategy pattern makes our software more resilient, and we can change its internal behavior by combining different objects. There is usually the most appropriate strategy object (algorithm) for each criterion, and we can specify the "best" algorithm while building a criterion at run time. For example, the minimum number of stations selects the BFS algorithm in our system, and the minimum traveling time criteria uses the famous Dijkstra algorithm.

As this pattern encapsulates a family of algorithms, different algorithms can be replaced with each other at any time, so if new algorithms are introduced in the future, we can also be flexible in responding to these changes.

4.3.3 Singleton Pattern

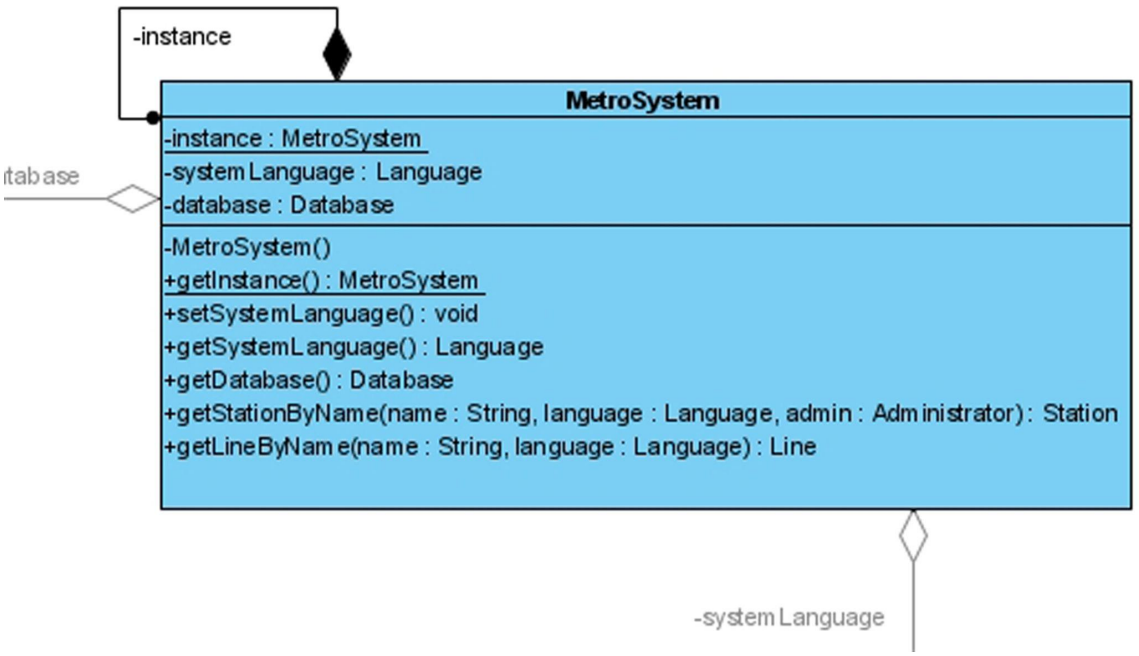


Figure 4.3.3 Example of Singleton Pattern - the Database Class

The Singleton Design Pattern ensures a class only has only one instance and provides a global point of access to it. [3]

The Singleton pattern is widely applied in our system design, such as Database and MetroSystem. For a class that will only have one object needed, to ensure no extra instance will be created that may cause information inconsistency and also reduce the system's occupation of computer resources thus improving overall performance at the same time, we apply the singleton pattern on it. Among all the classes, the most typical implementation is the database class, whose class diagram is shown above.

As for the detailed implementation of several key classes, given the JVM's multithreaded nature and modern operating systems' support for multithreaded programs, we paid special attention to the multi-threads performance issue on singleton pattern and applied double-checked locking method [4] when creating their instance.

4.3.4 Façade Pattern

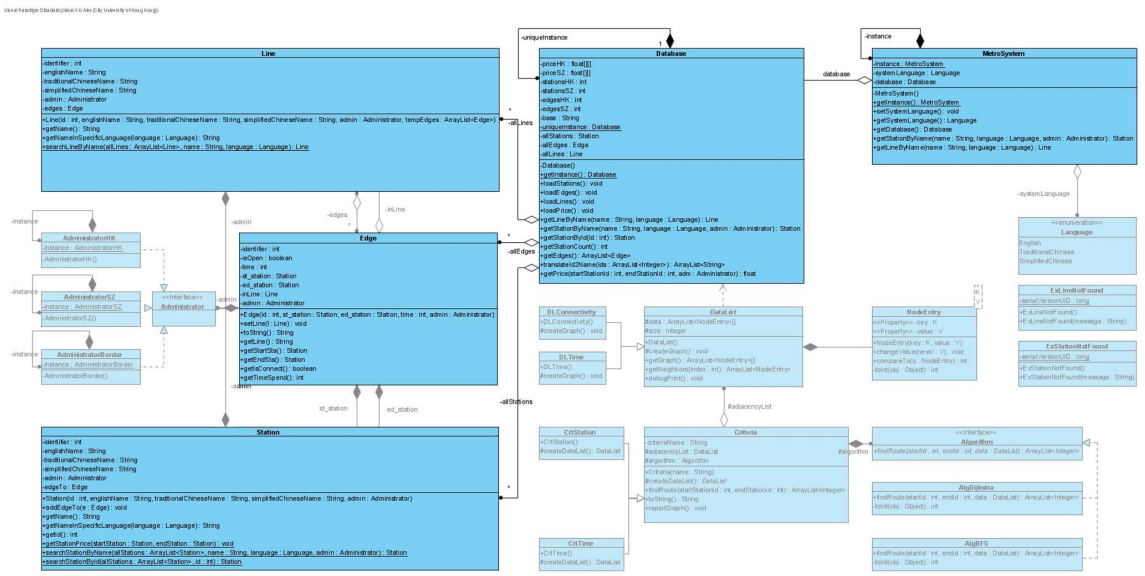


Figure 4.3.4 Example of Façade Pattern - the Database Class and Related Classes

The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. [3]

We define the database class as the Façade, which contains the set of simple functions that are made available to its client code. In detail, the Line, Station and Edge are all set to be the subsystem. The MetroSystem class is the higher interface(client) which tries to access the functions of subsystems.

By applying this pattern, the coupling between the sub system and the client is reduced, so that changes in the subsystem will not affect the client class that calls it. And the subsystem components are shielded from the customer, the number of objects processed by the customer is reduced. Consequently, it reduces the compilation dependency in large-scale software systems and simplifies the process of system transplantation between different platforms, because compiling one subsystem will not affect other subsystems, nor will it affect the appearance of objects.

4.3.5 Observer Pattern

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. [3]

We use SWING for GUI design. The Observer Pattern is regarded as the essence of the entire SWING. First, we need a container, JFrame, to display components. Then we create some components like panels, buttons, text areas and arrange the layout through the layout manager. The listener is attached to the button. It interacts with the component to generate an event, contact the user's actions. The addActionListener method is the subject's registered Observer Method.

This listener accepts action events as an excuse. The Main class, which is interested in handling action events, implements this interface. Objects created with that class are registered with the component using the addActionListener method of the component. When the action event is sent, the

execution of this object will be awakened. Frame display and all the dependents changed automatically.

4.3.6 Model-View-Control Pattern

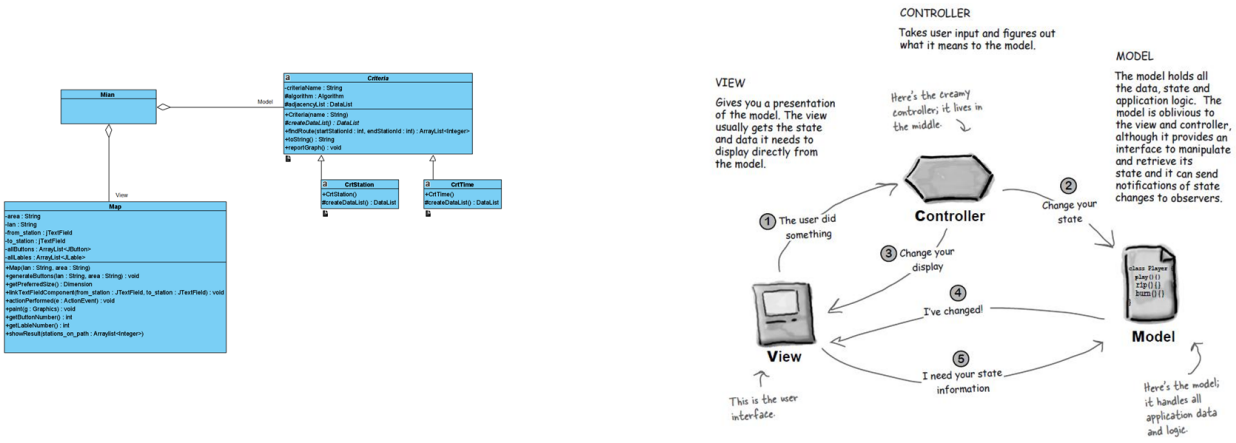


Figure 4.3.6 Example of Model-View-Control Pattern - Swing and UI Related Classes

The model-view-Controller Pattern is basic for our graphical user interface development tool: Swing. For example, The button can be in one of two states: enabled and disabled at any given moment. Obviously, the button will only respond to clicks when it is enabled. It is useful to record the state of the button. The view needs to be rendered differently, according to the state of the button. All the states of the button are the model part of the MVC architecture. The view is the graphical representation of the button. It uses the model to determine how to draw the button. Another important function of the button is that its state can be changed-when you click on it with the mouse, or press the space or enter key when it is in focus, the button can be activated. Obviously, there must be something to monitor how the mouse and keyboard and buttons get or lose focus in order to notice the necessary state changes. The part of the component responsible for receiving and responding to input is the controller.

5 PROGRAM FLOW AND ALGORITHMS

5.1 Overview

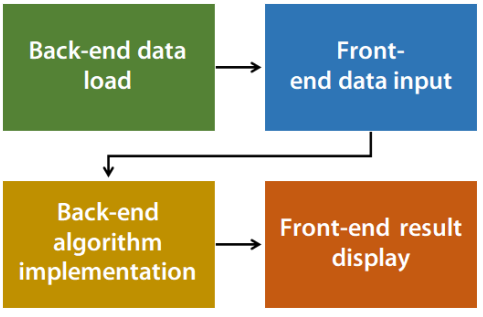


Figure 5.1 Program Flow of the System

The whole internal program flow procedure can be broken into four phases as shown in the above image.

About back-end data loading, all information about stations within different lines are all stored in back-end well-structured excel files. The system will automatically load the excel file using external library Apache POI to process data input, which makes our product easy to update when new metro lines are introduced in the Hong Kong or Shenzhen area. When first launching the program, the data will be loaded initially before users conduct path planning. Back-end database will store those information into three different categories, all stations, all edges connecting stations and all lines.

After data loaded and program initialization, a UI window will pop up. It allows users to do front-end data input. Users need to click the start stations and end stations in the map graph, which will invoke the related station data transferred to the back-end.

Back-end algorithm implementation is the core algorithm in our system. After this process, which will be discussed in detail later, the output data will be an arrayList containing all station ids along the shortest path.

Finally, the front-end UI will parse the information and highlight the shortest path using a bold red line in the UI window.

5.2 Sequence Diagrams

5.2.1 Core Algorithm

Our Metro system mainly uses two algorithms to realize the route selection for two different criteria. They are **Breadth-First Search**(aka. BFS) and **Dijkstra** respectively, while the corresponding criteria are the **least station number** base and **minimum time base**.

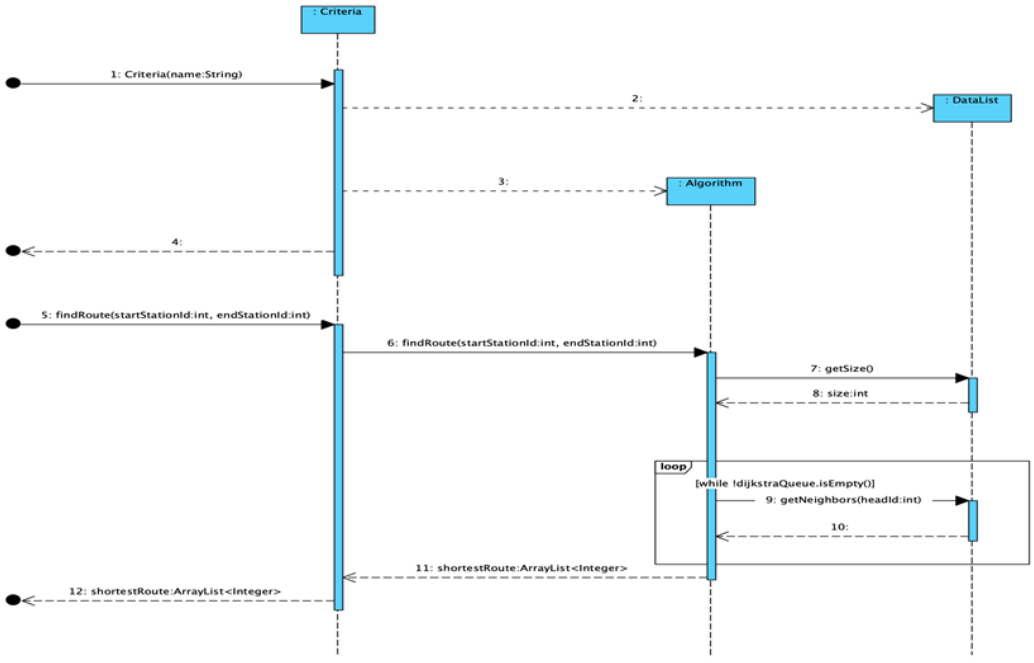


Figure 5.2.1 Sequence Diagram of core algorithm

5.2.2 Breadth-First Search

- Goodness of Fit

  As the metro system can be simply regarded as a graph with stations to be nodes and lines to be edges. When the start and the end stations are given, finding out the least station number route can be converted into generating the shortest path in an unweighted graph, which is a typical problem for the BFS algorithm.

- Code implementation

Data	**Usage**
ArrayList shortestRoute	Store the final result to be returned.
Queue bfsQueue	Store the nodes to be expanded sequentially.
int[] parent	Store the parent node id for each node that has been expanded for backtracking.
boolean[] visited	For identifying whether a node has been visited/expanded or not.
int headId	Current open node id.
int nextId	Neighbor node id of the current open node whether the singleton pattern of AdministratorBorder can work normally and prepare for future polymorphism use.

First initialize all stations to be unvisited except the start one, and push start station id into the bfsQueue. Then start the iteration. Every iteration checks one degree deeper for a current open node, then expands it if it is not the destination and is unvisited, and updates its parent at the same time. If the destination is reached, the iteration ends and backtracking starts.

- Sequence Diagram

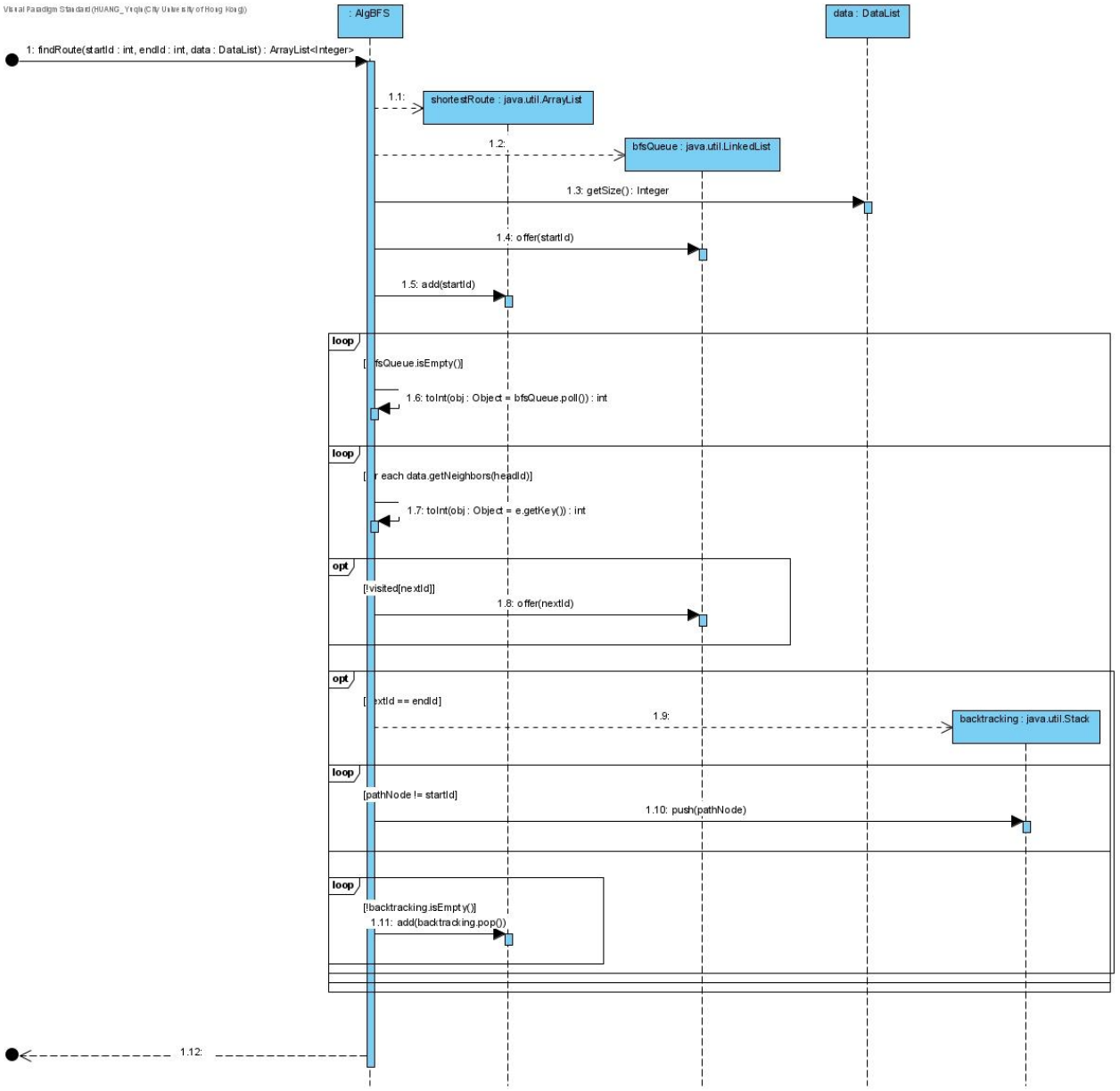


Figure 5.2.2 Sequence Diagram of core algorithm when using BFS

5.2.3 Dijkstra

- Goodness of Fit

  When it comes to the criterion of the minimum time taken, the graph becomes a weighted one, with the corresponding time to be spent being the value of each edge. Since Dijkstra is a more general algorithm compared to BFS, they are both used to solve the shortest path problems. The difference is that while the former can deal with weighted graphs, the latter can only be applied to the unweighted graphs. That's why Dijkstra can be perfectly used in this criterion.

- Code implementation

Data	**Usage**
ArrayList shortestRoute	Store the final result to be returned.
PriorityQueue dijkstraQueue	Store the nodes to be expanded in increasing priority order.
int[] parent	Store the parent node id for each node that has been expanded for backtracking.
int[] currentDis	Store the total distance between start station and current station, and at the same time mark the station as visited.
int[] toVisit	Store the shortest distance to an unvisited and to-be expanded value. To identify nodes that should be abandoned as it's not easy to remove nodes from the priority queue conditionally.
int headId	Current open node id.

The logic is similar to BFS, the difference is for Dijkstra, priority queue is used to choose the station which is the nearest to the start point in order to achieve the goal of least time spent.

- Sequence Diagram

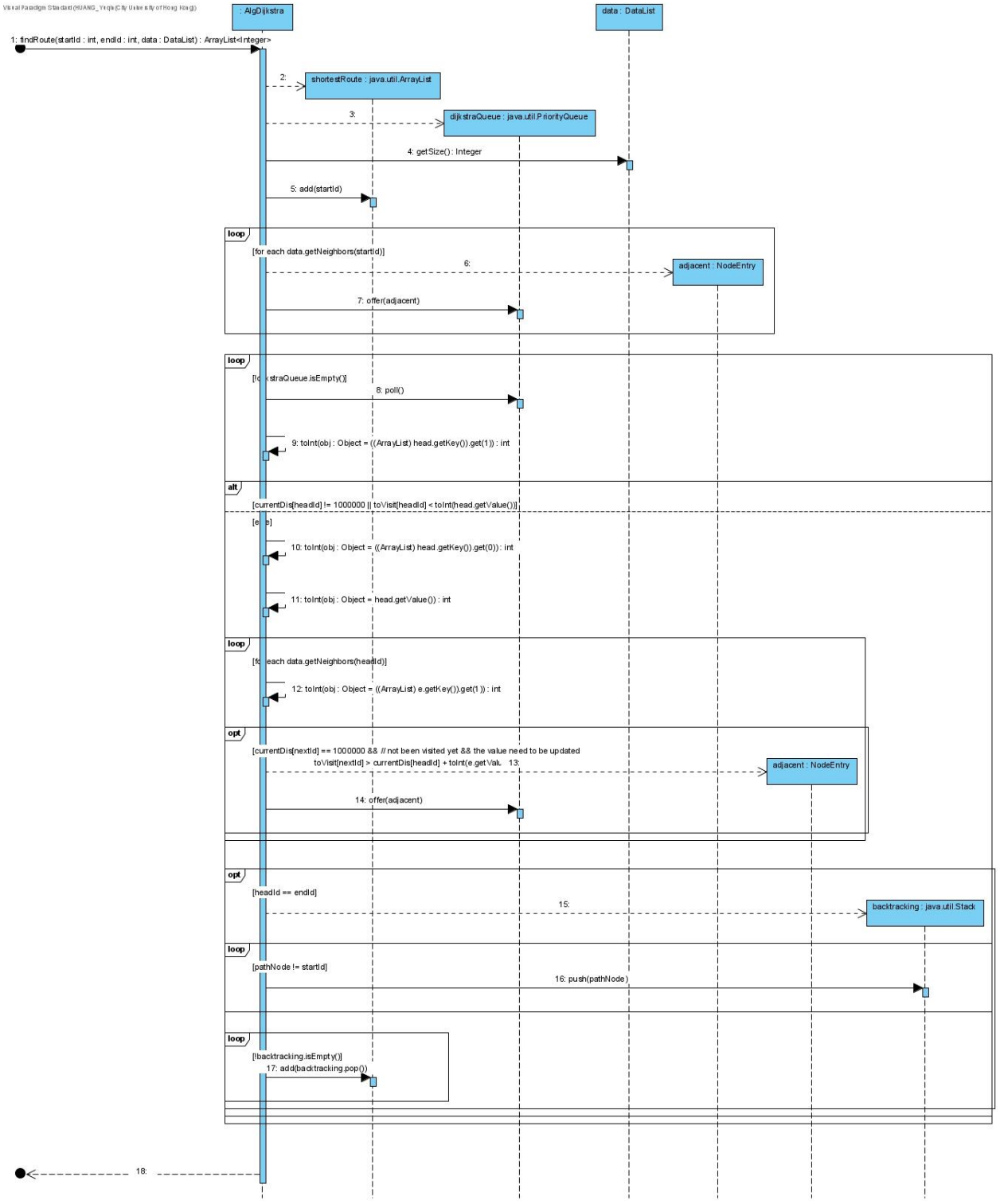


Figure 5.2.3 Sequence Diagram of core algorithm when using Dijkstra

REFERENCE

[1] T. DeMarco, *Structured analysis and system specification*. Englewood Cliffs, N.J.: Prentice-Hall, 1979.

[2] R. C. Martin, *Agile software development : principles, patterns, and practices*. Upper Saddle River, N.J.: Prentice Hall, 2003.

[3] E. Gamma, *Design patterns : elements of reusable object-oriented software*. Reading, Mass.: Addison-Wesley, 1995.

[4] Freeman et al., *Head First design patterns*, 1st edition. Sebastopol, California: O'Reilly, 2004.

[5] V. Sarcar, *Java Design Patterns*. Berkeley, CA: Apress L. P, 2018.

[6] Joydip Kanjilal, *Demystifying the Law of Demeter principle*, InfoWorld.com, 2016.