Lab 04: Exercising on Design Patterns

Aims:

1. Understand the concept and usages of several basic design patterns.
2. Programming in Java with design patterns.
3. Refactor a project to make it maintainable by applying various design patterns.

In the coursework, you are required to refactor a retro-game source code by implementing several design patterns (besides MVC pattern, which we will cover in the subsequent weeks). In this lab, we will practice on several basic and typical design patterns to help you forming a code sense of it.

The design patterns we will look at today include the following:

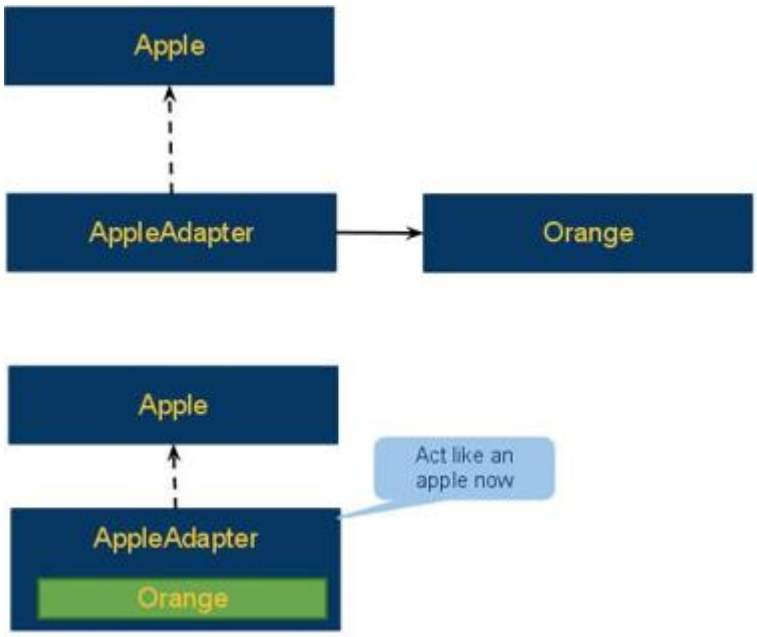
But please also exercise on the Strategy and Decorator patterns that are taught in the lecture, by understanding the code and looking at more examples online. For more design patterns and examples, refer to the various online resources , e.g., http://www.tutorialspoint.com/design_pattern/ or https://refactoring.guru/design-patterns

1. The Factory Method Pattern

Problem: An abstract ***Printer*** class is inherited by three subclasses, namely **GermanPrinter**, **FrenchPrinter**, and **EnglishPrinter**. The three printers are supposed to print only the language as specified in their names. The three printers are installed and used by FHSS and FoSE, where FoSE can only access the EnglishPrinter, and FHSS can access all the three printers. Design and implement the above requirement using the *Factory Method pattern*. You have the freedom to define necessary hierarchies, classes, variables, methods, etc.

2. The Adapter Pattern

The adapter pattern is used when you need to merge the functionalities of two less-related objects. An example is shown in the figure below, where we would like to merge **Orange** into **Apple**, say.



A general approach of merging **O** into **A** can be as follows:

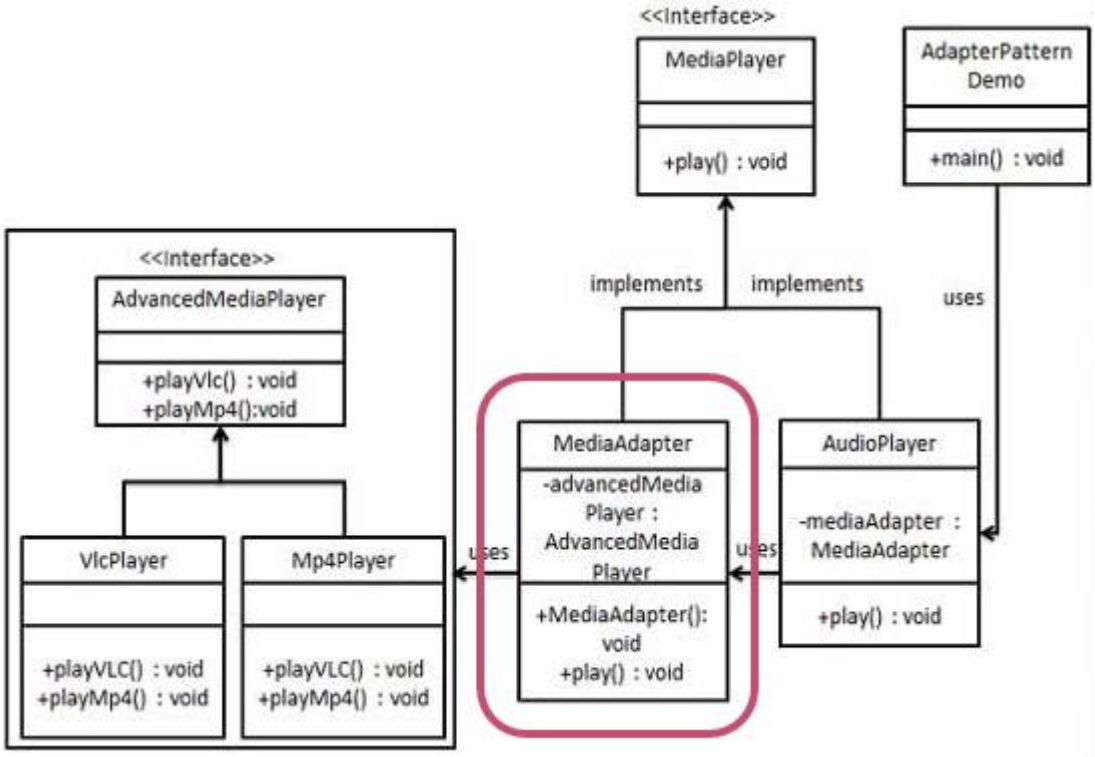
1. Create an **A_Adapter**, which is a kind of A or its inheriting classes.
2. This A_Adapter should contain an O object to behave O's functionality.
3. Then, put A_Adapter into the field of A, such that A can use the O object in the A_Adapter to behave O's functionality.

Another example: Let MP3 (audio player) to play the MP4/VLC (advanced media player) video, i.e., merge the advanced media player into MP3. Steps can be:

1. Create an audio adapter (also called media adapter in the figure below.).
2. Inside this adapter, maintain an advanced media object.
3. MP3 can then use the MP4 object in its audio adapter, to play MP4 videos.

A sample class diagram is shown below.

===
Task: Implement the class diagram below to see whether it works. Refer to sample code (a lighter version though) if necessary.
===



3. The Observer Pattern

The observer pattern describes a one-to-may relationships between the ***subject*** and its ***observers***, such that whenever the subject has been changed or modified, all of its observers get notified and updated automatically. Specifically, the actors in the observer pattern include two types:

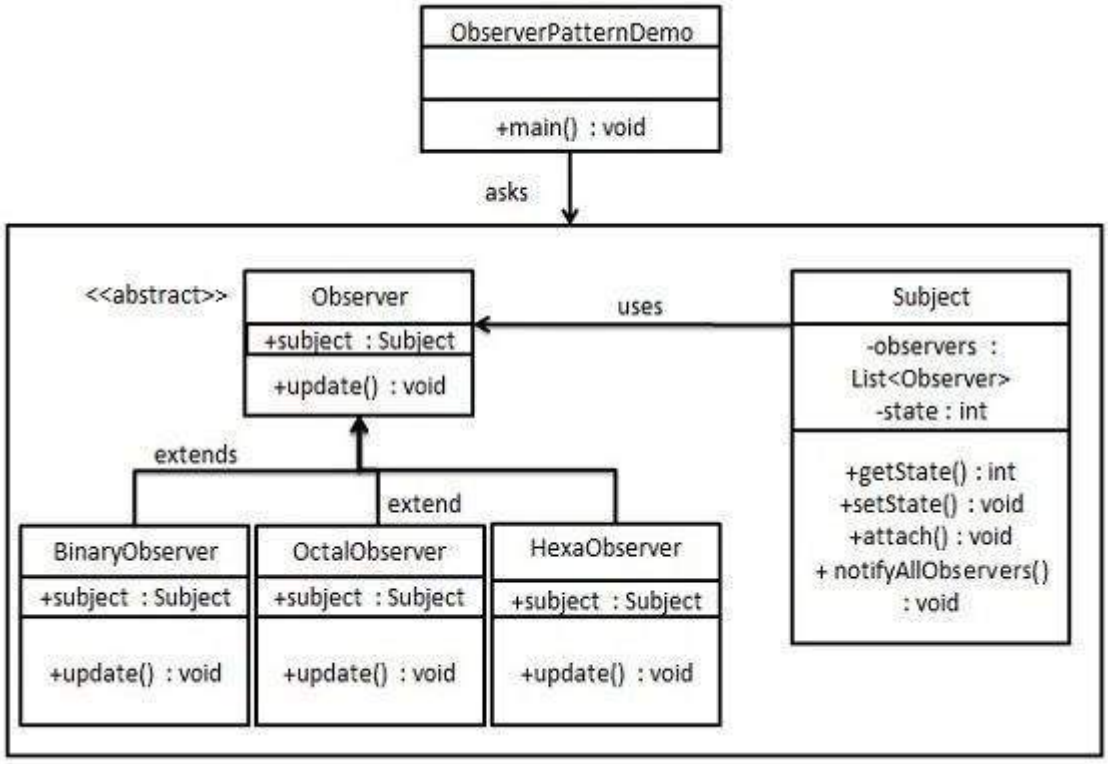
Subject: The object being watched. It should be able to attach/detach observers.
Observer/Listener: The objects watching or observing state changes. They register themselves with the Subject as they are created, and should contain some update methods in response to the changes of the Subject.

For example, you can define a listener for a button in an user interface. If the button is selected, the listener is notified and performs a certain action.

The operations of the observer pattern can in general be classified into:

- *Push model:* The Subject sends details to the observers regarding to what has been changed.
- *Pull model:* The Subject notifies the observers that something has been changed, the observers has to respond by querying from the subject about the changes. We focus on this model in our course.

An example class diagram is shown below. The Subject changes its state, which is an *int* number. Observers receives the changes, and print the *int* number in various formats.



The *attach(observer:Observer)* method registers an object in the List of the observer objects maintained in the Subject.

The *update()* method in Observers pull the new *int* state value of the Subject, and display accordingly.

The *subject:Subject* field is maintained in the Observer. Whenever an Observer object is initialized, in its constructor, it asks the *subject* to attach the observer itself into the observer list of the *subject*.

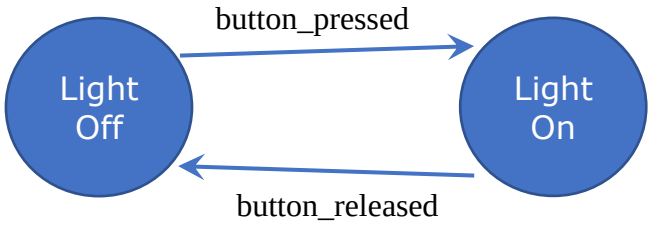
The *notifyAllObservers()* method is invoked whenever the observed value (in this example, the state *int*) is changed. So, where to place and invoke this method in the source code of the Subject class?

===
Task: Implement the above class diagram with the observer pattern. Refer to the sample code if necessary. To change the format of an integer, use the following clause, `Integer.toBinaryString(int)`, for conversion from int to binary.
===

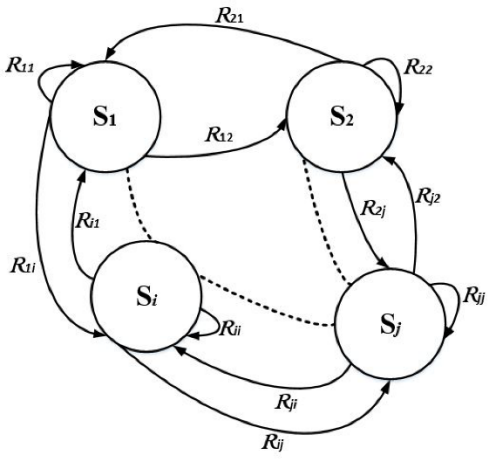
4. The State Pattern

A state machine is used to describe the behavior of an object *O*. It consists of node with directed edges. A node represents a state, and an edge represents an action that make *O* change its state from one to another. In the state machine below:

O is the Light;
States are {LightOn, LightOff}
Actions are {buttonpressed, buttonreleased}

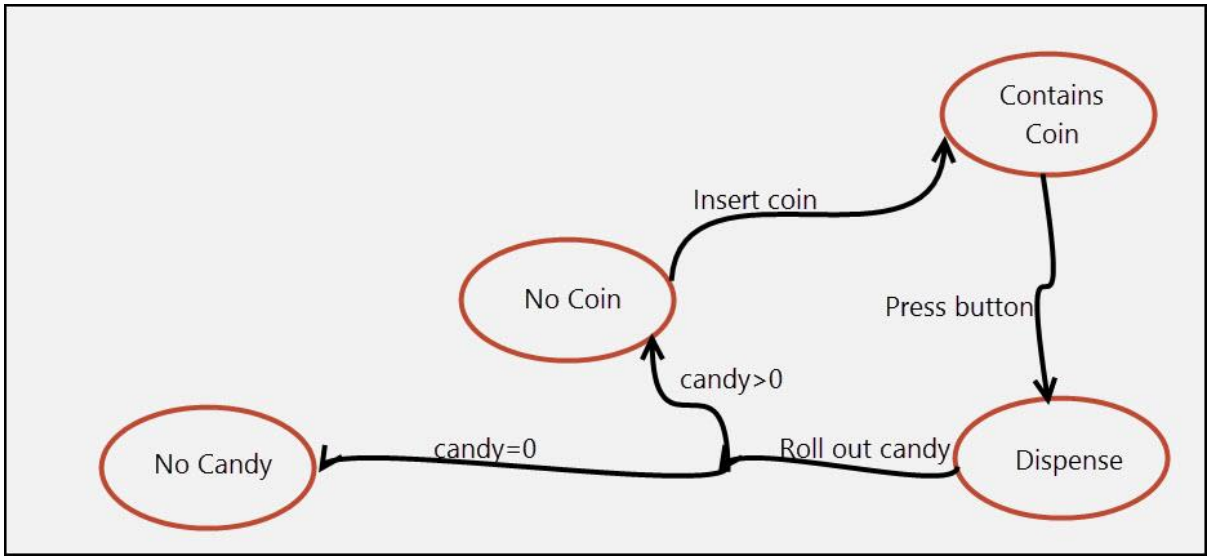


It can be more complicated as shown below.



The behavior of many real-world objects can be modeled by the state machine. What are the options to implement such behavior of the object in Java? One candidate is the `switch` (or `if-else`) statements to switch between different states and put the actions in each state inside the `switch` or `if-else` block. Another option is to adopt the State Pattern, which is a systematic and maintainable way to achieve the same goal.

Let's take a look at a vending machine example. The figure below shows its state diagram.



Firstly, study the implementation of the state diagram using if-else blocks. The code is shown below, though it would be a great practice if you try to work it out by yourselves.

```
public class CandyVendingMachine {
    final static int NO_CANDY = 0;
    final static int NO_COIN = 1;
    final static int CONTAINS_COIN = 2;
    final static int DISPENSE = 3;
    int count;
    int state = NO_CANDY;

    public CandyVendingMachine(int numberOfCandies) {
        count = numberOfCandies;
        if (count > 0) state = NO_COIN;
    }

    public void insertCoin() {
        if (state == CONTAINS_COIN) {
            System.out.println("Coin already inserted");
        } else if (state == NO_COIN) {
            state = CONTAINS_COIN;
        } else if (state == NO_CANDY) {
            System.out.println("No candies available");
        } else if (state == DISPENSE) {
            System.out.println("Error. System is currently
                                dispensing");
        }
    }

    public void pressButton() {
        if (state == CONTAINS_COIN) {
```

```
            state = DISPENSE;
        } else if (state == NO_COIN) {
            System.out.println("No coin inserted");
        } else if (state == NO_CANDY) {
            System.out.println("No candies available");
        } else if (state == DISPENSE) {
            System.out.println("No coin inserted");
        }
    }

    public void roll_out_candy() {
        if (state == CONTAINS_COIN) {
            System.out.println("No candies rolled out");
        } else if (state == NO_COIN) {
            System.out.println("No coin inserted");
        } else if (state == NO_CANDY) {
            System.out.println("No candies available");
        } else if (state == DISPENSE) {
            count = count - 1;
            if (count == 0) {
                state = NO_CANDY;
            } else
                state = NO_COIN;
        }
    }
}
```

In the code above, the actions (insert coin, press button, dispense) are defined in corresponding methods as the name indicates. The actions are basically the events that trigger the state transitions, represented as edges in the state diagram. In each action, two important behaviors should be noticed: 1) the state behaves accordingly, and 2) state transition happens.

The code appears concise and each to understand. However, in the maintenance point of view, it is unacceptable.

===

Think about it:

Imagining that a new state is added, and this state has edges to all other available states. Estimate the workload that incurred to change the above code.

- Four new methods should be added, corresponding to 4 new edges.

In each old method (`insertCoin`, `pressButton`, and `roll_out_candy`), code should be adjusted to insert the newly added state, namely, add statement for state transition.

- If the code was written by other programmer, you have to know all the details of the other states.
- All the above means huge workload to add a single state.
- Think about SOLID principles, the single-responsibility and open-closed principles are violated.

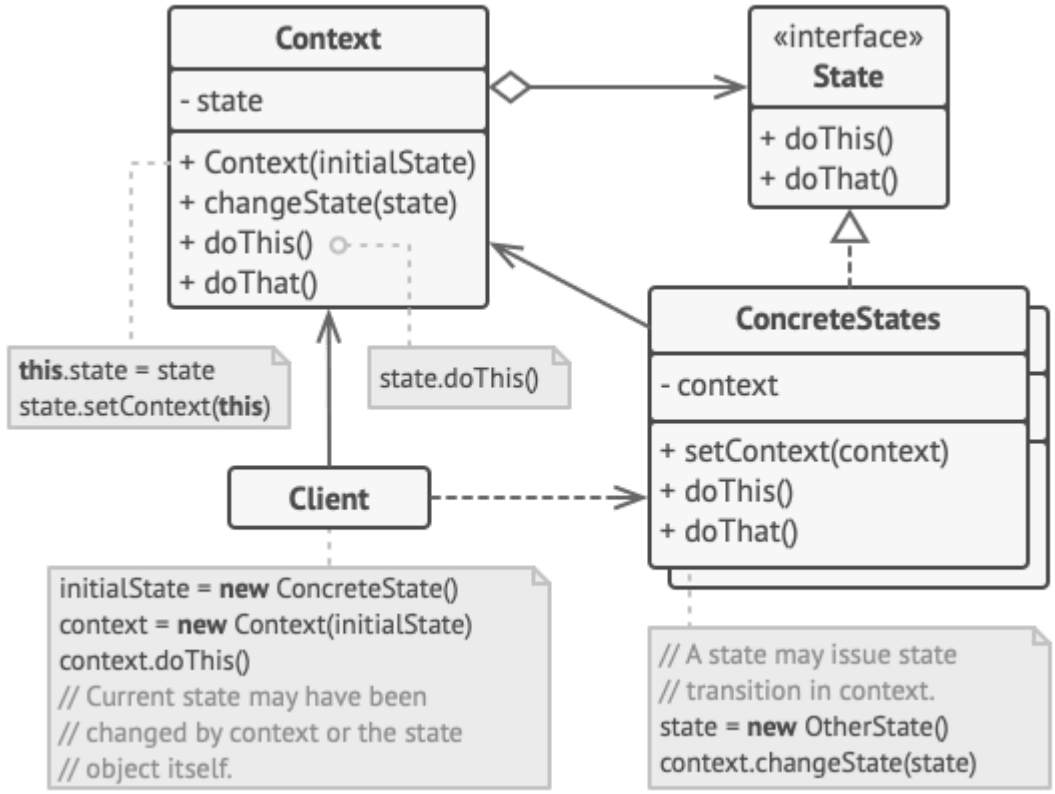
===

So much said, the state pattern nicely overcome the above issues.

Components of the state pattern:

- **State:** An <<Interface>> which declares what each concrete state class should do.
- **Concrete State:** Provides implementations for methods defined in the state interface. The methods should contain both actions in that state and state transitions.
- **Context:** Defines an interface that interacts with the client, which could be a *main()* method in a Tester class. It maintains references to concrete state objects which are used to define the current state of the object (vending machine).

The general structure of the State Pattern is shown below:



- The Context has many States rotating inside. The Context class has an abstract state variable inside to dynamically represent each concrete state. The `doThis()` method is a wrapper that wraps (calls) the `state.doThis()` method defined inside. The `state.doThis()` can be called because all ConcreteStates implement `doThis()` method – polymorphism.
- In each concrete state, the `state.doThis()` method contains the actions to be done in this state and the state transition from this state

after the actions finishes. The state transition is done by calling the `changeState` (or `setState`, etc.) method

We now apply the state pattern to the candy vending machine application. Let's start with the **State** interface.

```
public interface CandyVendingMachineState {
    void insertCoin();
    void pressButton();
    void roll_out_candy();
}
```

The state is an interface that defines all actions that should be acted by its concrete state classes. Note that it includes all actions of all concrete states, since all of them implement this interface.

Let's define a concrete state class, DispensedState, which corresponds to the "Dispense" state in the state diagram above.

```
public class DispensedState implements CandyVendingMachineState{
    CandyVendingMachine machine; //This is the Context.

    public DispensedState(CandyVendingMachine machine){
        this.machine=machine;
    }

    @Override
    public void insertCoin() {
        System.out.println("Error. System is currently dispensing");
    }

    @Override
    public void pressButton() {
        System.out.println("Error. System is currently dispensing");
    }

    @Override
    public void roll_out_candy() {
        if(machine.getCount()>0) {
            machine.setState(machine.getNoCoinState());
            machine.setCount(machine.getCount()-1);
        }
        else{
            System.out.println("No candies available");
            machine.setState(machine.getNoCandyState());
        }
    }

    @Override
```

```
    public String toString(){
        return "DispensedState";
    }
}
```

Notice that this state implements all the methods `insertCoin()`, `pressButton()`, and `roll_out_candy()`, although only the `roll_out_candy()` action is meaningful to this state (that is, has an output edge from this state). In this method, both state behaviors and state transitions are defined:

- State behavior: check candy count, if count > 0, count--, else, print info.
- State transition: check candy count, if count > 0, goto NoCoinState, else, goto NoCandyState.

Also observe that this class contains a field called '`machine` : **CandyVendingMachine**', which is the **Context** of the State Pattern. Recall the definition: "It maintains references to concrete state objects which are used to define the current state of the object (vending machine)".

In the code above, the concrete state objects are objects of the NoCoinState and NoCandyState, obtained from `machine.getNoCoinState()` and `machine.getNoCandyState()`.

Let's define the CandyVendingMachine class.

```
public class CandyVendingMachine {
    CandyVendingMachineState noCoinState;
    CandyVendingMachineState noCandyState;
    CandyVendingMachineState dispensedState;
    CandyVendingMachineState containsCoinState;
    CandyVendingMachineState state;
    int count;

    public CandyVendingMachine(int numberOfCandies){
        count=numberOfCandies;
        noCoinState=new NoCoinState(this);
        noCandyState=new NoCandyState(this);
        dispensedState=new DispensedState(this);
        containsCoinState=new ContainsCoinState(this);
        state = noCoinState;
    }

    public void clientInsertCoin(){
        System.out.println("You inserted a coin.");
        state.insertCoin();
    }
    public void clientPressButton(){
        System.out.println("You have pressed the button.");
        state.pressButton();
        state.roll_out_candy();
```

```
    }

    /* Getters and Setters of the local variables/states. */
}
```

Observe how the concrete state objects are defined and initialized in this Context.

Also note that the `clientInsertCoin()` and `clientPressButton()` methods defined in Context are called by the client (in other words, the outside world, the Tester class, etc), to let the Context to complete the actions desired by the client.

===
Task: Implement the other three concrete state classes, namely, **NoCoinState**, **NoCandyState**, and **ContainsCoinState**. Also implement a **Tester** class to verify the functionality of this implementation.
===

===
Think about it:
What should be changed to add another state into the application? Would the implementation and logic of other concrete states be affected? Compare to the previous if-else implementation, is this State pattern more maintainable?
===

Refer to https://refactoring.guru/design-patterns/state
or https://springframework.guru/gang-of-four-design-patterns/state-pattern/ for more details and explanations.

5. The Singleton Pattern

As one of the simplest design patterns, the Singleton pattern ensures a class that it instantiates into only one object at runtime.

Implementation tips:

- Make the constructor private,
- Define a static private instance of itself and let the client use it.

```
public class SingleObject {
    //create an object of SingleObject
    private static SingleObject instance = new SingleObject();

    //make the constructor private so that this class cannot
    // be instantiated
    private SingleObject(){}
```

```
    //Get the only object available
    public static SingleObject getInstance(){
        return instance;
    }

    public void showMessage(){
        System.out.println("Hello World!");
    }
}


public class SingletonTester {
    public static void main(String[] args) {
        //illegal construct
        //Compile Time Error: The constructor SingleObject()
        // is not visible
        //SingleObject object = new SingleObject();

        //Get the only object available
        SingleObject object = SingleObject.getInstance();

        //show the message
        object.showMessage();
    }
}
```

Refer to the sample code provided in Moodle for a comparison.

Now, with your knowledge on all the design patterns learnt so far, apply them on the FleetManagement system implemented in the previous week's lab.

Applying design patterns to FleetManagement

1. Factory Method pattern

Consider to use a factory method pattern to create the lorry/ship/train objects. If the text file for carrier initialization contains the following lines shown in Fig. 3, then using a carrier factory would make the `DataManager.loadCarriers()` method more maintainable – less if-else logic in the DataManager method. But it leaves the complexity to the factory class – single responsibility. Note that this factory method pattern is not a complete version that has been described in page 42 of Lecture 4 notes, but more similar to the one shown in page 41. This is because we are not given more information to construct the full factory method pattern. The aim of this exercise is only for you to practice creating a factory class containing a createCarrier() method inside.

```
Lorry 4
Ship 1
Lorry 6
Train 2
Lorry 8
Lorry 10

```
Fig. 3. Format of the text file containing carriers' information.

2. State pattern

Consider to realize a state pattern for the lorry states. The general rule of implementing the state pattern is to create a sub-class for each individual state. The state transitions are defined as methods in the parent state class. Review Lab03 on the state pattern section to understand the advantages of the state pattern.

3. Observer pattern

Implement a RoutePlannerObserver, which is notified if the route is changed. Recall the `routeChanged(routeInstance, routeNodeInstance)` method in the RoutePlanner class. Perhaps this is where you need inset the notification statements. Define the RoutePlannerObserser class, which usually contains an `update()` method, to react to the change of the observed value. The RoutePlanner should now be added with an additional field, typically named observers, which are RoutePlannerObserver objects. Also, the RoutePlanner should now define an additional method named `notifyObservers()` to inform the observers. Review Lab03 on the observer pattern section if necessary.

4. Singleton pattern

Refactor the DataManager class to be a singleton, such that it has one and only one instance in the system at any time. Recall how to make a class singleton: make the constructor private, and define a private static DataManager object, named instance, which is typically accessible by a static `getInstance()` method. Note how outside classes/objects access this singleton class and its methods.

===

Task:

- Revise the class diagram below, found in Week 3's lab, to reflect the modifications on applying design patterns.
- Continue working on the fleet management code, and refactor the project using the four design patterns mentioned above. Refer to the sample code [FleetManagement_Patterns.zip] for your reference.
- Other important design patterns taught, including the Adapter, Decorator, and Strategy patterns, are not covered in this example. Think about new scenarios in this example that can be suitable to apply those design patterns.

===

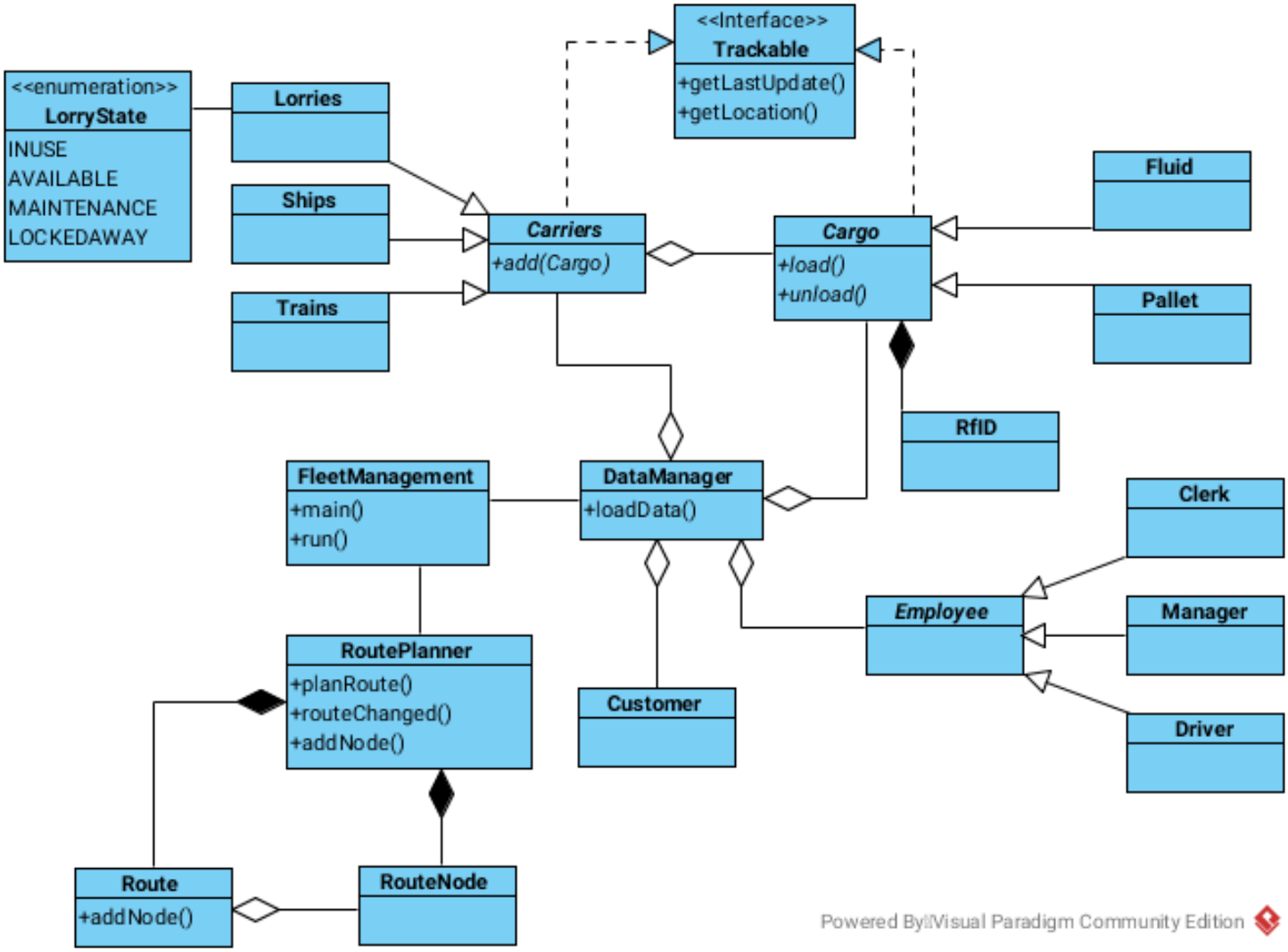


Fig. 1. Sample class diagram of the FleetManagement_Basic project.