Task 1:

(I): Using an observer design pattern sets up a one-to many dependency so when an object representing internal state changes, other objects (such as display objects) can be notified and update themselves accordingly.

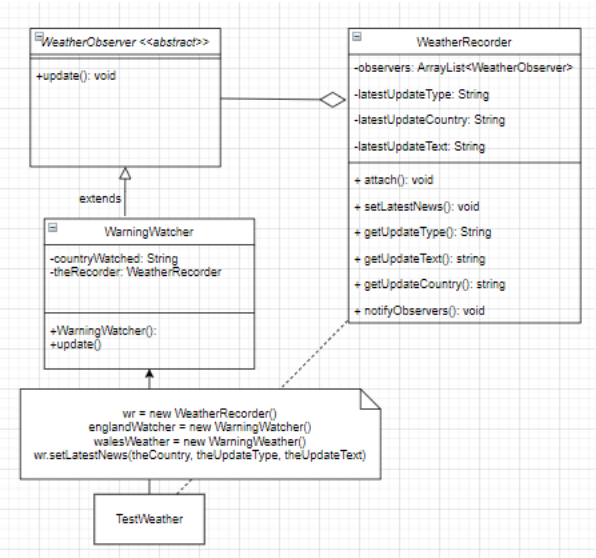
(ii): WeatherObserver is an abstract update() stub so only information inherited from WarningWatcher (since this class extends from WeatherObserver) will be accessed and show the information to the user. Using this abstract ensures only one interface can be adapted to all classes so WeatherRecorder will not couple to concrete classes and declaring a notification interface.

In TestWeather, there is main(string args) which is the main class to initiate the whole programme and linking classes, whether locally on the same file or in different Java files in the same or linking directory. This class creates other objects seperately then subscribes them into the WarningRecorder class to record the warning update.

For WarningWatcher, it is a class which the eponymous public class sets a specific wr from TestWeather.java and sets it as the private WeatherRecorder class called theRecorder, noting the warning message attributed to the country, which is specified from TestWeather.java and applied to the private string countryWatched. The update function in this class allows the output for the whole program, informing if the statement of theRecorder.getUpdateType() matches the word “Warning”.

In WeatherRecorder, it collects the ArrayList observers from the WeatherObserver so that it is compatible with the class WarningWatcher, which is added to the list of observers through the add module in attach function. It also has setLatestNews function to collect the Country, Update type and Update Text. Then it calls the private function notifyObservers to Iterate through the list. This class permits subscriber messages to update the weather status, whether giving information on a country’s forecast or warning status.

The UML Diagram:



(iii): WeatherRecorder attach()

```
public void attach(WeatherObserver o) {
// Complete this method so that it adds the observer to the list of observers
}
```

Becomes

```
public void attach(WeatherObserver o) {
// Complete this method so that it adds the observer to the list of observers
  observers.add(o);
}
```

WeatherRecorder notifyObservers()

```
  private void notifyObservers() {
// Complete this method to go through each observer in turn,
// sending it a message to notify that an update has occurred
  }
```

becomes

```
private void notifyObservers() {
  Iterator <WeatherObserver> iterator = observers.iterator();
  while (iterator.hasNext()) iterator.next().update();
Complete this method to go through each observer in turn,
sending it a message to notify that an update has occurred
}
```

WarningWatcher update()

```
public void update() {
// Modify this so that it only prints out the update text if the update is
    System.out.println("The WarningWatcher watching for Warnings for " +
                       countryWatched +
                       "\nhas noticed a new warning:\n\"" +
                       theRecorder.getUpdateText() + "\"\n");
}
```

becomes

```
public void update() {
// Modify this so that it only prints out the update text if the update is a "Warning" for the countr
  String updateStatus = theRecorder.getUpdateType();
  if (updateStatus.equals("Warning") && this.countryWatched.equals(theRecorder.getUpdateCountry())){
  System.out.println("The WarningWatcher watching for Warnings for " +
                     countryWatched +
                     "\nhas noticed a new warning:\n\"" +
                     theRecorder.getUpdateText() + "\"\n");
}
```

(iv): My minor changes mean that instead of no output, the programme now outputs what is stated in the question, it contributes by implementing a while loop in the notifyObserver function whether there is hasNext for the iterator. However, in contrast to the example I was given for the WarningWatcher update(), I had to constrain the iteration with &&, so that only Wales gets the ‘Very Windy’ warning and England gets the ‘Very Snowy’ Warning, not both countries. That way, the getUpdateCountry() only applies to the respective countryWatched().

Task 2:

(I): Using a strategy design pattern allows the definition of a family of algorithms, all invoked in the same way so each algorithm can vary independently of clients that invoke it.

(ii): Added Code:

```
public class MostFuturisticStrategy implements ChoiceStrategy {
    public Product chooseBetween(Product a, Product b){
        int AFuture = a.futuristicness;
        int BFuture= b.futuristicness;
        if (AFuture >= BFuture) {
            return a;
        } else{
            return b;
        }
    }
     // Complete this with an implementation of
    // chooseBetween(Product a, Product b) which returns
    // Product a if its futuristicness is greater than or equal
    // to that of b; and returns Product b otherwise
}
```

(iii): Added Code:

```java
public class MostPracticalStrategy implements ChoiceStrategy {
    public Product chooseBetween(Product a, Product b){
        int APractical = a.practicality;
        int BPractical= b.practicality;
        if (APractical >= BPractical) {
            return a;
        } else{
            return b;
        }
    }
    // Complete this with an implementation of
    // chooseBetween(Product a, Product b) which returns
    // Product a if its practicality is greater than or equal
    // to that of b; and returns Product b otherwise
}
```

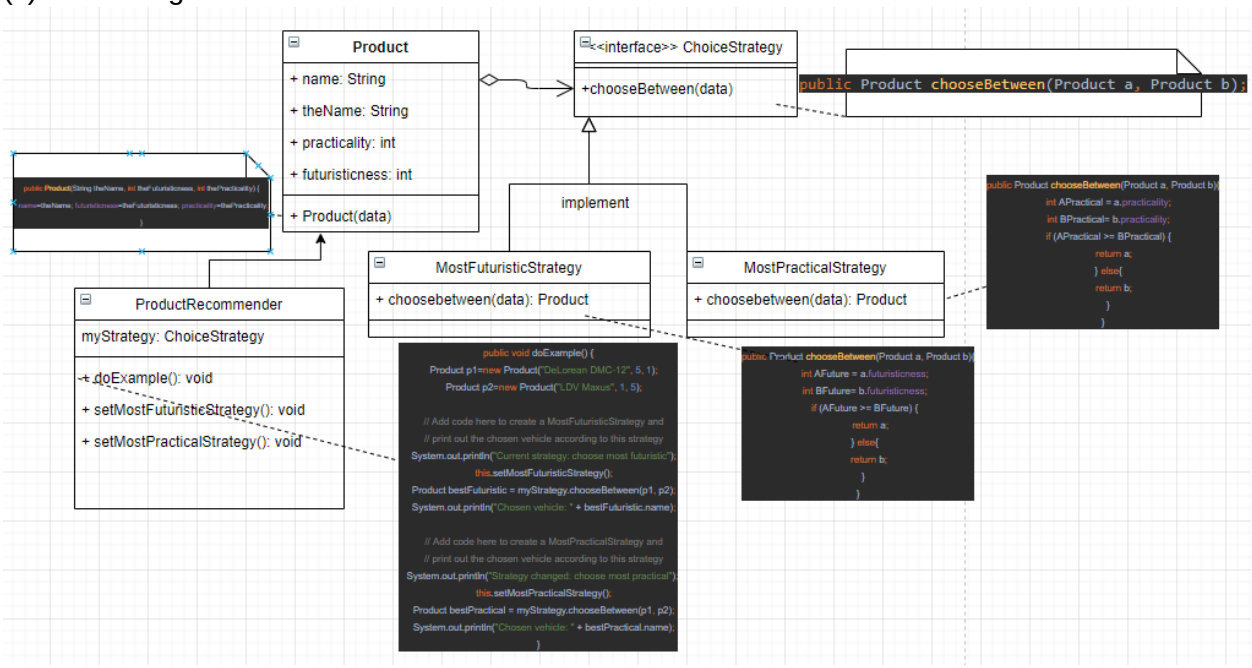
(iv): Added Code:

```java
public class ProductRecommender {

    ChoiceStrategy myStrategy;

    public static void main(String args[]) {
        ProductRecommender recommender=new ProductRecommender();
        recommender.doExample();
    }

    public void setMostFuturisticStrategy() { myStrategy = new MostFuturisticStrategy(); }
    public void setMostPracticalStrategy() { myStrategy = new MostPracticalStrategy(); }

    public void doExample() {
        Product p1=new Product( theName: "DeLorean DMC-12",  theFuturisticness: 5,  thePracticality: 1);
        Product p2=new Product( theName: "LDV Maxus",  theFuturisticness: 1,  thePracticality: 5);

        // Add code here to create a MostFuturisticStrategy and
        // print out the chosen vehicle according to this strategy
        System.out.println("Current strategy: choose most futuristic");
        this.setMostFuturisticStrategy();
        Product bestFuturistic = myStrategy.chooseBetween(p1, p2);
        System.out.println("Chosen vehicle: " + bestFuturistic.name);

        // Add code here to create a MostPracticalStrategy and
        // print out the chosen vehicle according to this strategy
        System.out.println("Strategy changed: choose most practical");
        this.setMostPracticalStrategy();
        Product bestPractical = myStrategy.chooseBetween(p1, p2);
        System.out.println("Chosen vehicle: " + bestPractical.name);
    }
}
```

output:

```
"F:\Program Files\Java\jdk-16.0.1\bin\java.exe" "-javaagent:F:\Prog
Current strategy: choose most futuristic
Chosen vehicle: DeLorean DMC-12
Strategy changed: choose most practical
Chosen vehicle: LDV Maxus

Process finished with exit code 0
```

(v): UML Diagram:



(vi): The role of ChoiceStrategy interface allows the ProductRecommender class to utilise the interface so that all classes implementing this interface ensure not only the support from the two Strategy Classes (I.e. MostFuturisticStrategy and MostPracticalStrategy), but also disable this interface from modifying. This ensures only methods in the interface will be runnable from implemented classes; any other methods in the extended classes but not in the interface, will make the whole programme gain errors. Without this interface, the whole programme crashes when a class tries to pull a method whose parameters are incompatible within the interface format.

For the two strategy classes, after receiving the client request and data (I.e. car model, and its futuristicness and practicality), they compile and run their if statements, then they return them to the client as the most futuristic and most practical, respectively.

The Product class basically initiates as a gateway and adaptor for the ProductRecommender client Class to adapt the interface. It does this by taking the called method with the respective parameters and their format and value and duplicates them into interface-supported format data. It maintains a reference to the two strategy classes, commuting this with just the ChoiceStrategy interface.

The ProductRecommender sets the Strategy format into the ChoiceStrategy interface, then initiates the doExample() method to create new Product dataset, which lets the client replace the strategy associated with the Product class when the programme runs. It also initiates the setMostFuturisticStrategy() and setMostPracticalStrategy() methods, which ensures the support for the two strategy classes to compare and return the result of their respective highest calibre.

Task 3:

(i): Added CounterClass.java to the directory consisting of:

```
public class CounterClass implements Runnable{
  private int value = 0;
  public synchronized int getNext() { return value++; }
  public int getResult() { return value; }
  public void run() {
    for (int i=0;i<5000;i++){
      this.getNext();
    }
  }
}
```

Output:

```
Result of doTask1: 10000
```

(ii): doTask2() turns out to do the exact same thing as doTask1(), as it applies the same Class as doTask1() returning the Runnable formatted in a loop up to 5000 using this.getNext() for t1, then applying the same for t2, starting them and joining them for 10000 in get result. doTask2() is a failsafe to ensure the threads are not different and does exactly as what doTask1() does. Sometimes, without a failsafe, the method, while doing the same operations, may return different results than expected, depending on the model, Operating System and the hardware used or not even execute at all.

Output:

```
Result of doTask2: 10000
```