Formating Instructions: Please use this provided LaTeX template to complete your homework. When including figures such as UMLs, it is highly encouraged to use tools (graphviz, drawio, tikz, etc..). Figures may be hand draw, however, these may not receive credit if the grader cannot read it. For ease of grading, when including code please have it as part of the same pdf as the question while also including correct formatting/indent, preferably syntax highlighting. Latex includes the minted, or lstlisting package as a helpful tool. For this assignment all code must be full code (no pseudo-code) and be written in either Java or C++. However, implements may ignore all logic not relevant to the design pattern with simple print out statements of "[BLANK] logic done here"

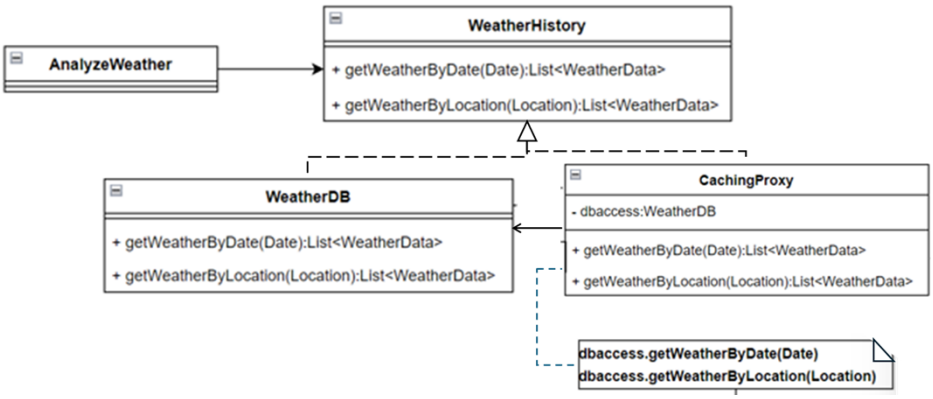
Question Instructions: In this homework assignment, you will apply the Structural patterns discussed in the lectures.
This is an individual assignment, you may not work in groups
For each question:

1. Give the name of the design pattern(s) you are applying to the problem.
2. Present your reasons why this pattern will solve the problem. Please be specific to the problem and do not give general applicability statements. If there is an alternative pattern, explain why you preferred this one..
3. Show you design with a UML class diagram. If the pattern collaborations would be more visible with another diagram (e.g. sequence diagram), give that diagram as well.
   (a) Your diagram should show every participant in the pattern including the pattern related methods.
   (b) In pattern related classes, give the member (method and attribute) names that play a role in the pattern and effected by the pattern. Optionally, include the member names mentioned in the question. You are encouraged to omit the other methods and fields.
   (c) For the non-pattern related classes, you are not expected to give detailed class names etc. You may give a high-level component, like "UserInterface" or "DBManagement"
4. Give Java or C++ code for your design showing how you have implemented the pattern.
   (a) Pattern related methods and attributes should appear in the code
   (b) Client usage of the pattern should appear in the code
   (c) Non-pattern related parts of the methods could be a simple print. (e.g. "System.out.println()", "cout")
5. Evaluate your design with respect to SOLID principles. Each principle should be addressed, if a principle is not applicable to the current pattern, say so.

1. (26 points) We are developing a software application for an energy company, and we need to access the weather history data from a database. We make queries by date or by location. An example weather data is: "Golden CO, November 10th, 2021, daytime, average temperature 54°F, 57% Humidity". This information does not change once it is inserted into the weather history database.

   (a) (11 points) For performance considerations, we do not want to access the database each time we need the weather data in our application. Since the weather data does not change once it enters the history, we could use caching (preloading the data) and return the cashed values whenever weather information is requested instead of database access. We access the database only when the requested data is not in the cache. Whether the data is cached or not should be a concern for the rest of the application. Suggest a structural design pattern for this purpose. (address all items 1-5)

   1. Proxy (caching proxy)
   2. The problem is about controlling access to the database. We want to access the database only if the data has not been asked for before. Using proxy, the actual data access happens only when needed. Client code should not be aware of whether it is a cached value or not. Proxy satisfies this constraint as well since it has the same interface as the actual access.
   3. UML



   4. Code

```
/* Subject interface*/
public interface WeatherHistory{ //some sensible name.
     public List<WeatherData> getWeatherByDate (Date date);
     public List<WeatherData> getWeatherByLocation(Location location);
}
/*Proxy*/
public class CachingProxy implements WeatherHistory{
     private WeatherDB dbaccess;
     public CachingProxy(WeatherDB db){dbaccess=db;}
     private Collection< WeatherData> cached=new ArrayList<WeatherData>();
        //any collection type is ok
        //actually use Map for efficiency.
```

```
    public List<WeatherData> getWeatherByDate(Date date){
       /* example code. My code is not the most efficient.
        * Point is: if the data is not found in cached,
        * make the actual call to dbaccess.
        * Then, save those data in the cache.
        * Return the result of the query.
       */
        List<WeatherData> result=new ArrayList<WeatherData>();
        for(WeatherData w: cached){
          if(w.date().equals(date)) result.add(w);
        }
        if(result.isEmpty()){
          result=dbaccess.getWeatherByDate(date);
          cached.add(result);
        }
        return result;
      }
      public List<WeatherData> getWeatherByLocation(Location location);
        List<WeatherData> result=new ArrayList<WeatherData>();
        for(WeatherData w: cached){
            if(w.location().equals(date)) result.add(w);
        }
        if(result.isEmpty()){
            result=dbaccess.getWeatherByLocation(location);
            cached.add(result);
        }
        return result;
      }
}
/* RealSubject*/
public class WeatherDB implements WeatherHistory{
  public List< WeatherData> getWeatherByLocation(Location location){
    System.out.println("making SQL query using the date"); //this is sufficient
    return null;
    //returning some data since this code is a prototype
    //Non-pattern related parts of the methods could be a simple print.
  }
     //alternatively, a more detailed code could be given.
    // here is an example inside the getWeatherByDate implementation.
    //some database connection code like JDBC
   private Connection connection;
   private void makeConnection() throws Exception{
   System.out.println("making connection");
   connection=DriverManager.getConnection(null,null,null); //not necessary.
    //only a Simple print is ok. 2nd line is not necessary.
   }
```

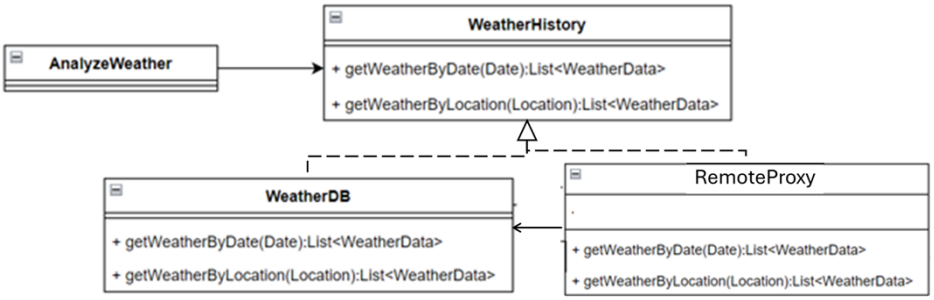
```
    public List<WeatherData> getWeatherByDate(Date date){
         System.out.println("making SQL query using the date"); //this is sufficient
         List <WeatherData> result=new ArrayList<WeatherData>();
        //if you want more detail...
        //...but this code is not necessary for our purposes
       try{
         Statement statement=connection.createStatement();
          String sql; //fill in the SQL to query by date
         ResultSet r=statement.executeQuery(sql);
                //...process the result set and make a WeatherData object list
        }catch (Exception e){e.printStackTrace();}
       //just returning a dummy weather data, not a pattern related code
      result.add(new  WeatherData(null,date,null,0.0,0.0));
      return result;
    }
}
public class WeatherData{
    //NEVER use public for attributes!
    private final Location location;
    private final Date date;
    private final String timeOfDay;
    private final float temperature;
    private final float humidity;
    public WeatherData(Location location, Date date, String timeOfDay,
              float meanTemperature, float humidity){
       this.location=location; this.date=date;
       this.timeOfDay=timeOfDay;
       temperature=meanTemperature; this.humidity=humidity;
    }
  //getter methods as well. No setters. In C++ this is easier.
}
class Location{ private String city; private String state; /*...*/}

//Client
public class AnalyzeWeather{ //some sensible class name.
    private WeatherHistory history;
    public AnalyzeWeather(WeatherHistory historyDB){
         history=historyDB;}
    public void someAnalysis(){
         Date date=new Date("November",10,2021);
         List<WeatherData> data=history.getWeatherByDate(date);
         /*...do analyze*/
    }
}
public class TestAnalyze{
  public static void main(){
```

```
        WeatherHistory h=new CachingProxy(new WeatherDB());
        AnalyzeWeather analysis=new AnalyzeWeather(h);
       h.analyze();
}
}
```

5. SOLID
   - Single responsibility: Proxy is responsible for caching the results and making the database access if there is a change miss. WeatherDB is only responsible for database operations.
   - Open-closed: Open to new caching logic with new proxy implementations.
   - Liskov Sub.: Client code works as expected both with the proxy and with the actual object.
   - Interface segregation: Not directly relevant to this pattern
   - Dependency inversion: Both the proxy and the WeatherDB depend on the same interface, promoting flexibility and decoupling. Analysis is dependent on the WeatherHistory interface. So, high level depends on high level.

(b) (7 points) We have switched to a remote weather database. Currently, the remote connection, sending and receiving data using HTTP or Rest API request are all over the codebase. Suggest a structural pattern so that the rest of the application is not dependent on such remote procedures. (address items 1-4)

1. Remote Proxy
2. The complexity of Rest API or HTTP client code will pollute the energy part of the application. We could isolate the application from these remote operation details with a remote proxy that has the same interface as the weather database service. The application will access Weather history data no different than accessing any other local object
3. UML



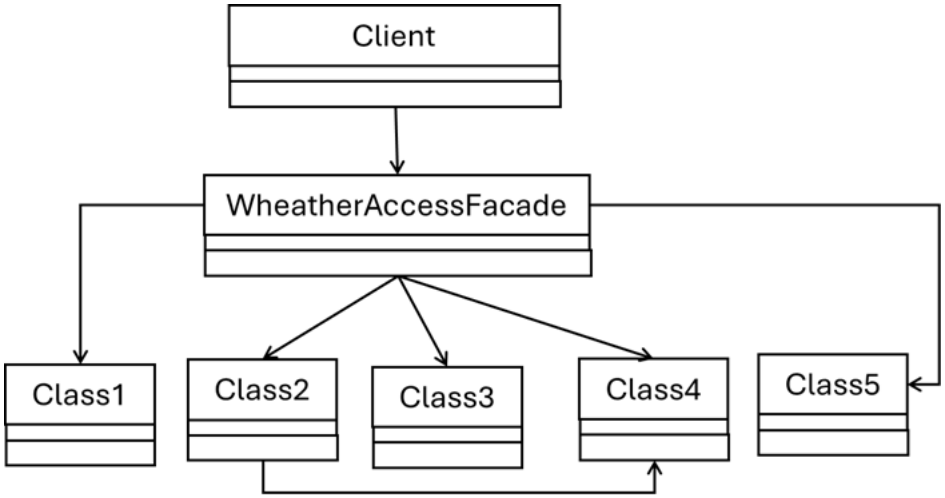
4. Code

```
1  public class WeatherData{ /* same as part A */}
2  public class WeatherDB implements WeatherHistory{
3          /*same as part A*/}
4  public interface WeatherHistory{ /*same as part A*/}
5    /*However, we could change the interface to throw Exception
6    * or RemoteException at each method. i
7    * Since they are remote operations, something may probably go wrong.
8    * On the other hand, if we want to keep the analyze component
9    * (i.e. the rest of the application) exactly the same,
10   * we way consider to handle those exceptions inside the proxy.
11   */
12 public class RemoteProxy implements WeatherHistory{
13   private WeatherDB dbaccess;
14   public RemoteProxy(WeatherDB db){
15     System.out.println("setting up the rest API client");
16     System.out.println("making a remote connection via http or rest");
17     dbaccess=db;  //simulating the remote connection
18   }
19   /* here we have a choice. We could cache the results as before,
20    * or we could simply relay the call to the remote API.
21    * Both are ok.
22    */
23   public List<WeatherData> getWeatherByDate(Date date){
```

```
24        List<WeatherData> result=new ArrayList<WeatherData>();
25        try{
26            System.out.println("making a rest API call using date");
27           /*simulating the result*/
28           result.add(new  WeatherData(null,date,null,0.0,0.0));
29        }catch(Exception e){
30            System.out.println("when remote exception occurs, handle it
              ↪  based on the requirements. Maybe try it 2-3 times, maybe
              ↪  return an emptylist");
31            return result;
32       }
33       public List<WeatherData> getWeatherByLocation(Location location){
34          System.out.println("making a rest API call using location");
35          return new ArrayList<WeatherData>();
36
37       }
38   }
39   //client
40   public class AnalyzeWeather{ /* same as part A*/}
41   public class TestRemoteAnalyze{
42     public static void main(){
43          WeatherHistory h=new RemoteProxy(new WeatherDB());
44          AnalyzeWeather analysis=new AnalyzeWeather(h);
45          h.analyze();
46     }
47   }
```

(c) (8 points) I want to restructure my code. Currently there are five interconnected classes that are responsible for weather data access and processing. I want to give simplified access to these from my energy related components. (You may assume any name for the five interconnected classes). Suggest a structural design pattern that would simplify this and discuss advantages and disadvantages of your design. (address items 1,2,3,5)

1. Facade
2. The energy related code has to know all of the five interconnected classes and there will be many dependencies from energy code group to data access code group. Because of the interconnection, it could be complex and error prone to directly access them. Façade will provide a simplified interface for the energy code group so that: a) direct dependencies will reduce b) code becomes more understandable and maintainable (any change in the data access/process code will be handled in the façade to reduce the impact of the change in the energy code) c) complexities because of interconnectedness will be hidden inside the façade.
3. UML



some interconnections between classes

5. SOLID
   - Single responsibility: The facade has a single responsibility of providing a simplified interface. Underlying classes are responsible for data access and processing. This separation ensures that changes in the subsystem do not affect the facade, and vice versa.
   - Open-closed: The facade can be extended with new methods without modifying existing ones. New facades could be added without affecting the rest. If the underlying classes change, the facade will change but the rest of the application may not be affected.
   - Liskov Sub.: Façade is not directly related to LSP
   - Interface segregation: The facade exposes only the necessary methods, ensuring clients depend only on what they use and need.

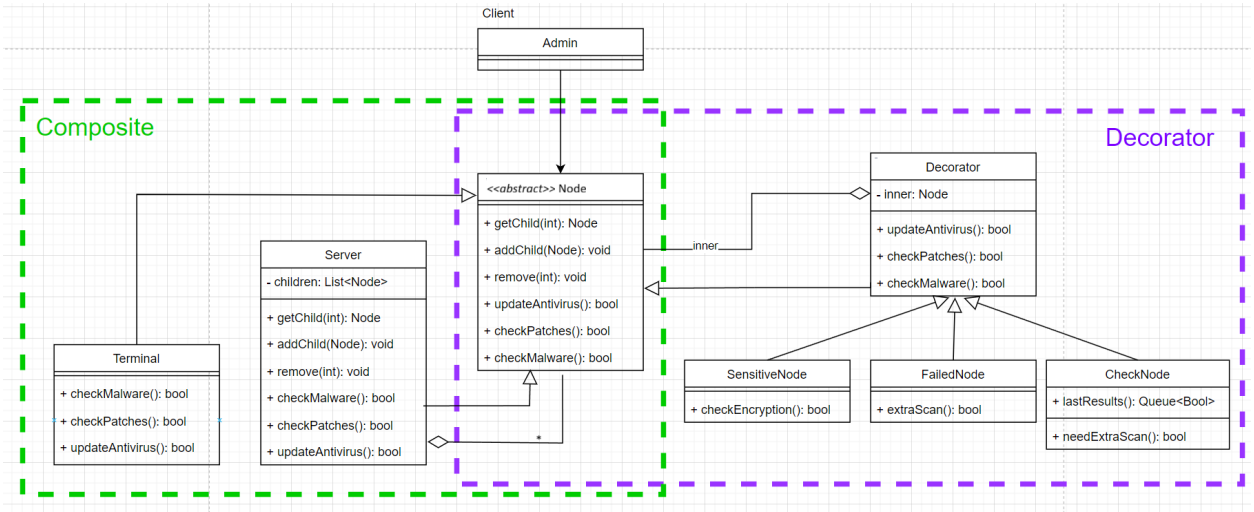
- Dependency inversion: Façade is the abstraction. High level components interact the data processing through the façade. It decouples high level components from low-level subsystem.

2. (15 points) (15 points) Your team has designed a system to manage the security of a hierarchical network of computers, implemented using the Composite pattern. The network consists of a main server (a composite node) connected to several sub-servers (composite nodes), each connected to multiple client computers (leaf nodes). The system supports the following requirements:

   Security Checks: Each component (main server, sub-server, or client) supports basic security checks: scanning for malware, updating antivirus software, and applying security patches. As an admin, you can initiate these checks at any component (e.g., the main server for the entire network, a sub-server for its sub-network, or a single client). Dynamic Additional Checks: Certain components require additional security measures based on specific conditions: Servers handling sensitive data must perform an additional encryption check. Any component that has failed any of the last five security checks requires an extra deep security scan. If a component passes five consecutive checks, this scan is removed. (Note: This is a simplified scenario for educational purposes.)

   Flexibility: The system must support adding new types of security checks and new component types in the future without modifying existing code. Given that the network hierarchy is implemented using the Composite pattern, design a solution using a structural design pattern to handle the dynamic addition and removal of security checks as described. Justify your choice of pattern, explaining how it complements the Composite structure and meets the flexibility requirements. (address all items 1-5)

   1. Composite (basic security checks) and Decorator (dynamic additional checks)
   2. The composite pattern will allow the admin to run basic security checks on all components in a uniform fashion. The Decorator will allow the addition of additional checks on any component in a dynamic fashion.
   3. UML:



   4. Code:

```
1 /*component interface*/
2 public abstract class Node{
```

```
3       public Node getChild(int i){
4           return null;/*default behavior for leaf*/     }
5       public boolean addChild(Node n){ return false; /*default behavior*/}
6       public Node removeChild(int n){ return null; /*default behavior*/}
7       public abstract boolean checkMalware();
8       public abstract boolean updateAntivirus();
9       public abstract boolean checkSecurityPatches ();
10      /* ALTERNATIVE:
11       * having only one security check method that handles all checks*/
12  }
13
14  /* Node could also be an interface with
15   * no default behavior implementation*/
16
17  /*leaf node*/
18  public class Terminal extends Node{
19    private int id;
20    public class Terminal(int ID) {this.id=ID;}
21    public boolean checkMalware(){
22      System.out.println(
23          "checking for malware at the client computer "+id);
24      return true;
25    }
26    public boolean updateAntivirus(){
27      System.out.println(
28          "updating the antivirus software at the client computer "+id);
29      return true;
30    }
31    public boolean checkSecurityPatches (){
32      System.out.println(
33          "ensuring that all security patches are up-to-date at the client
            ↪  computer" +id);
34      return true;
35    }
36  }
37
38  /*composite node*/
39  public class Server extends Node{
40       private int id;
41       public class Server(int ID) {this.id=ID;}
42       //must implement the child management operations
43       protected List<Node> children=new List<Node>();
44       public Node getChild(int i){
45          if(i>=0 && i<children.size()) return children.get(i);
46          return null;
47       }
```

```
48      public boolean addChild(Node n){
49         children.add(n);
50         return true;
51      }
52      public Node removeChild(int i){
53         if(i>=0 && i<children.size()) return children.remove(i);
54         return null;
55      }
56
57      public boolean checkMalware(){
58         System.out.println("checking for malware at the server "+id);
59         boolean result=true;
60         for(int i=0;i<children.size();i++){
61             Node node=children.get(i);
62             if (!node.checkMalware()) {
63                 System.out.println("Malware check failed "+ i + "th
                    ↪  branch of the server "+id);
64                 result=false;
65             }
66         }
67         return result;
68       }
69       public boolean updateAntivirus(){
70         System.out.println(
71             "updating the antivirus software at the server "+id);
72         boolean result=true;
73         for(Node node:children){
74             result=result && node.updateAntivirus();
75         }
76         return result;
77       }
78       public boolean checkSecurityPatches (){
79         System.out.println("ensuring that all security patches are
            ↪  up-to-date at the server "+id);
80         boolean result=true;
81         children.forEach(node ->
82             {result=result &&  node. checkSecurityPatches()});
83         return result;
84       }
85  }
86
87  /*Component interface is the same as Question5. It does not change*/
88  /* Abstract decorator,
89   * so that the concreate decorators do not have to
90   * implement every abstract method
91   */
```

```
public class Decorator extends Node{
      protected Node inner;
      public Decorator(Node inner){this.inner=inner;}
      public boolean checkMalware(){ return inner.checkMalware();}
      public boolean updateAntivirus(){ return inner.updateAntivirus();}
      public boolean checkSecurityPatches (){
              return inner.checkSecurityPatches();}
}
/*concrete decorator */
public class SensitiveNode extends Decorator{
       public boolean checkEncryption() {
             System.out.println("checking encryption");
       }
}

/*concrete decorator*/

public class CheckCounting extends Decorator{
    private Queue<Boolean> lastresults = new ArrayQueque<>(5);
    private void recordResult(boolean result){
      if(lastresults.size()>=5) lastresults.remove();
            lastresults.add(result);
      }
    public boolean checkSecurityMalware(){
      boolean result= inner.checkMalware();
      recordResult(result);
      return result;
    }
    public boolean updateAntivirus(){
      boolean result= inner.updateAntivirus();
      recordResult(result);
      return result;
    }
    public boolean checkSecurityPatches (){
      boolean result=  inner.checkSecurityPatches();
      recordResult(result);
      return result;
    }
    public boolean needExtraCheck(){
      if(lastresults.size()<5) return false;
      for( boolean r: lastresults) if(!r) return true;
      return false;
      }
    }
/*concrete decorator*/
public class FailedNode extends Decorator{
```

```
138     public boolean extraScan(){
139             boolean result= inner.checkMalware() ;
140             result=result && inner.updateAntivirus();
141             result=result && inner.checkSecurityPatches();
142             return result;
143     }
144     public Node getInnerNode(){ return inner;}
145 }
146
147 public class Admin{
148    private Node root;
149    public buildNetwork(){
150         root=new CheckCounting (new Server(1));
151         root.add(new CheckCounting (new Terminal(2)));
152         Server server= new CheckCounting (new Server(3));
153         server.add(new CheckCounting (new Server(4)));
154         server.add(new CheckCounting (new Terminal(5)));
155         root.add(new SensitiveNode(server));
156
157         root.checkMalware();
158     }
159  }
160
161
162
```

5. SOLID:
   - Single responsibility: The composite separates the managing of the hierarchy structure and the individual leaf responsibilities. The Decorator separates tasks into individual dynamically linkable responsibilities.
   - Open-closed: New leaf nodes and decorators can be created without any modification to existing code.
   - Liskov Substitution: The admin can treat the Terminal and Server children of the Node type the same, they are able to be replaceable for one another.
   - Interface segregation:
   - Dependency inversion: The client is able to interface with the high level Node class which decouples the client from any details in any of the concrete lower level classes.

Table 1: Grading Rubric

Grading Rubric for Part A - use the grading rubric for 10 points questions		
Grading Rubric for Part B - 7 points		
1 (1 point)	0	missing or incorrect
	+1	correct pattern
2 (1 point)	0	missing
	+1	the reason provided correctly describes an advantage of the pattern and is specifically beneficial to this scenario
3 (2 points)	0	missing
	+1	includes all participants (including client) that play a role in the pattern
	+1	all class relations are correct
4 (3 points)	0	missing
	+1	includes all pattern related methods and attributes
	+1	includes client usage
	+1	correctly implements and uses all pattern related methods
Grading Rubric for Part C - 8 points		
1 (1 point)	0	missing or incorrect
	+1	correct pattern
2 (2 point)	0	missing
	+1	the reason provided correctly describes an advantage of the pattern and is specifically beneficial to this scenario
3 (3 points)	0	missing
	+1	includes all participants (including client) that play a role in the pattern
	+1	all class relations are correct
	+1	includes all class members that are related to the pattern
5 (2 points)	0	missing
	+2	correctly lists multiple ways the pattern benefits a user

Table 2: Grading Rubric for **10** points questions

1 (1 point)	0	missing or incorrect
	+1	correct pattern
2 (1 point)	0	missing
	+1	the reason provided correctly describes an advantage of the pattern and is specifically beneficial to this scenario
3 (3 points)	0	missing
	+1	includes all participants (including client) that play a role in the pattern
	+1	all class relations are correct
	+1	includes all class members that are related to the pattern
4 (3 points)	0	missing
	+1	includes all pattern related methods and attributes
	+1	includes client usage
	+1	correctly implements and uses all pattern related methods
5 (2 points)	0	missing
	+2	correctly lists multiple ways the pattern benefits a user

Table 3: Grading Rubric for **15** points questions

1 (1 point)	0	missing or incorrect
	+1	correct pattern
2 (1 point)	0	missing
	+1	the reason provided correctly describes an advantage of the pattern and is specifically beneficial to this scenario
3 (6 points)	0	missing
	+2	includes all participants (including client) that play a role in the pattern
	+2	all class relations are correct
	+2	includes all class members that are related to the pattern
4 (5 points)	0	missing
	+1	includes all pattern related methods and attributes
	+2	includes client usage
	+2	correctly implements and uses all pattern related methods
5 (2 points)	0	missing
	+2	correctly lists multiple ways the pattern benefits a user