Lab 8

	Student Name	Student CSUSM ID	Contribution percentage
1	EJ Lilagan	lilag002	25%
2	Neo Argatides	argat001	25%
3	Gabriela Rojas	rojas124	25%
4	Benno Wiedner	wiedn002	25%

100

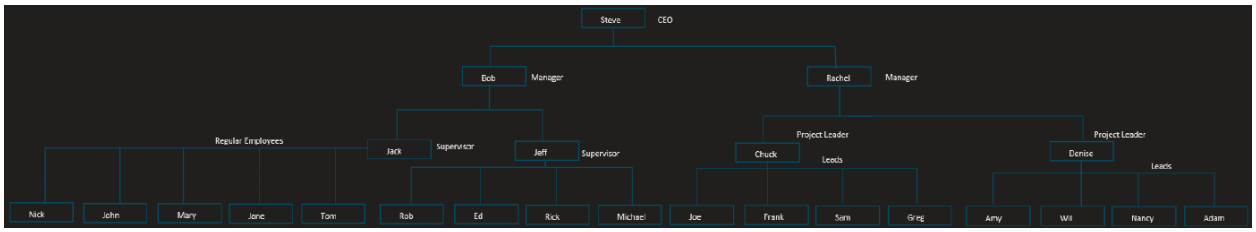
Grading Rubrics (for instructor only):

Criteria	1. Beginning	2. Developing	3. Proficient	4. Exemplary
Modeling	0-14	15-19	20-24	25-30
Program: functionality *correctness*	0-9	10-14	15-19	20
Program: functionality *Behavior Testing*	0-9	10-14	15-19	20
Program: quality -> *Readability*	0-2	3-5	6-9	10
Program: quality -> *Modularity*	0-2	3-5	6-9	10
Program: quality -> *Simplicity*	0-2	3-5	6-9	10
Total Grade (100)				

Problems:

Company XYZ has a hazard control model. There are two supervisors Jack and Jeff, and two project leaders Chuck and Denise. Jack supervises regular employees (workers) John, Mary, Jane, Tom and Nick. Jeff supervises Rob, Ed, Rick and Michael. Chuck leads Joe, Frank, Sam

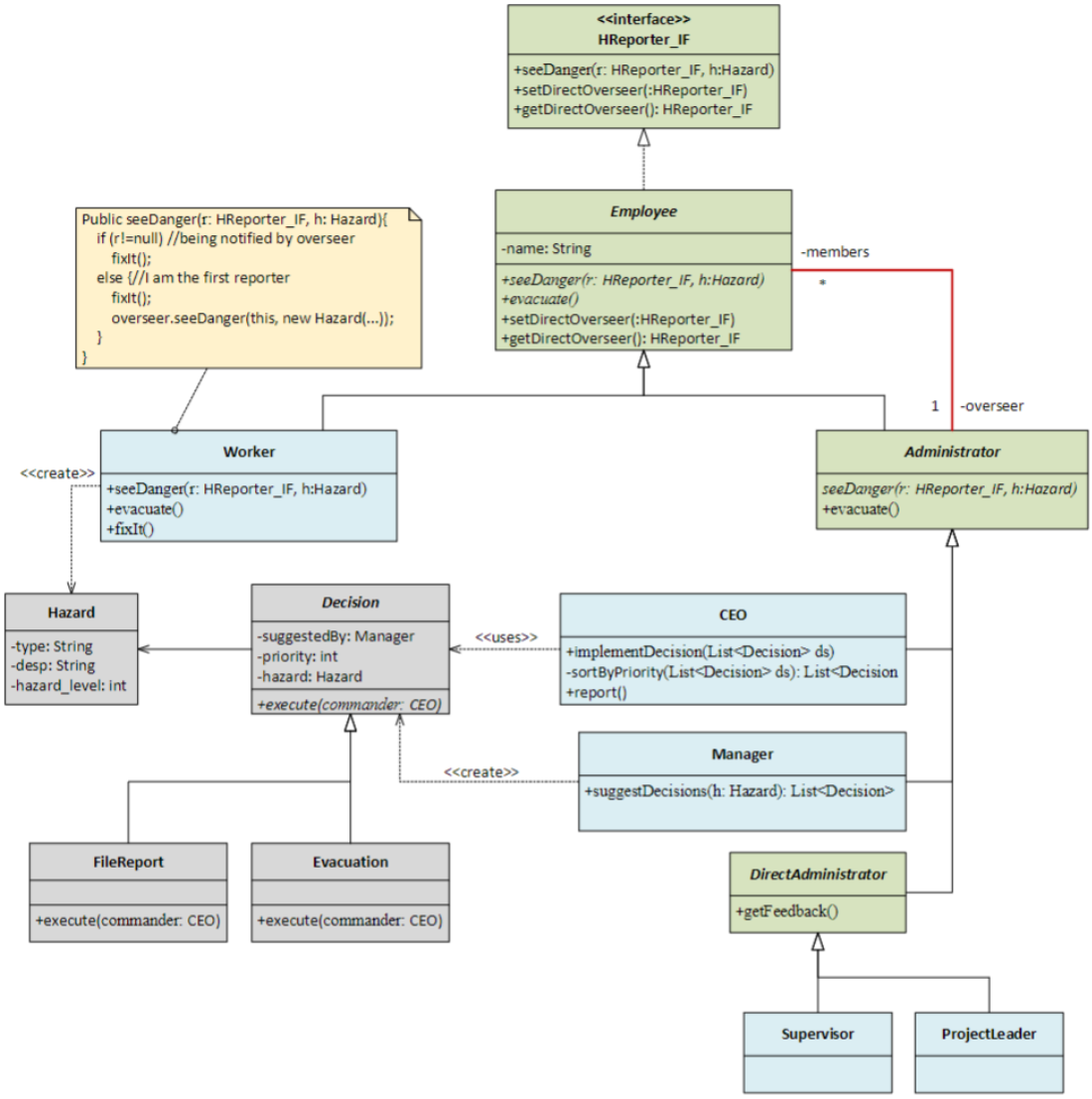
and Greg. Denise leads Amy, Wil, Nancy and Adam. Bob is the manager of Jack and Jeff, while Rachel is the manager of Chuck and Denise. The CEO is Steve. Their relations are shown below.



When a regular employee (worker) identifies a danger, the issue is reported to his/her supervisor or project leader. The supervisor or leader will announce the trouble to all regular employees in his/her team and also report it to the manager in charge. The manager will collect feedbacks from his/her managed supervisors or leaders and contact the CEO if necessary. The CEO will collect decisions suggested by all managers. Eventually, the CEO pick the final decisions.

Below is an architecture design of the system.

SE 471 Software Architecture



Here is a scenario. A worker John observed a gas leak of a big tank and triggered method "void seeDanger()" to report it to his supervisor. newThe supervisor ran "void seeDanger()" to tell all his team members to perform fixIt() and also inform his manager. The fixIt() method prints out a message like "The employee [name] is fixing it." The manager ran "void seeDanger()" to handle the danger by asking feedbacks from all supervisors/leaders under his management and contacting the CEO in case the feedbacks are all positive (true). Each supervisor or leader object has a "boolean getFeedback()" method, displaying "Feedback by [name]" and returning true if the hazard needs to be reported to upper level of administration. The CEO ran "void seeDanger()" to collect suggested decisions from the managers who performed their "suggestDecisions(h: Hazard): List<Decision>" method. The CEO makes up his final decisions by method "implementDecision(List<Decision> ds)".

After sorting all the decisions that he received by priorities, assume that the CEO always chooses to implement the first two decisions. Each decision has a method execute (:CEO). Assume there are two types of decisions. The first decision is of type Evacuation and the second decision is of type FileReport. The execute () of a decision is always initiated by the CEO directly. The execute () of the FileReport decision simply displays "The city's environmental department is notified". The execute () of the Evacuation decision demands all employees in the company to evacuate. The evacuation execution must start with an administrator's subordinates first then the administrator him-/herself (in particular, the CEO is always the last one to evacuate). When a person's evacuate() method is called it displays "The employee [name] is evacuating".

Solution:

- First, remember to zip the src folder of your project and submit the zip file to the ungraded assignment named "Lab8CodeSubmission". One submission from each team.
- Please explain the design pattern(s) being used in the design.
- Implement the design in Java. Paste a screenshot of a run of your program here.
- Also paste all you source code here.
- Save this report in PDF, and submit the pdf report to the graded assignment named "Lab8ReportSubmission". One submission from each team.

The design pattern being used in the design is Chain of Responsibility. Chain of Responsibility lets you pass requests along a chain of handlers and in this design, a worker sees danger and begins escalating the hazard notification up the chain of roles. At every level, there is a different process which is defined in seeDanger which determines the next level of escalation. The last level, CEO, makes the decision on how to handle the danger escalated to them.

```
A worker John observed a gas leak of a big tank and triggered method to report it to his supervisor

The employee John is fixing it.
Notifying direct overseer...

Instructing all team members to fix it...

The employee Nick is fixing it.

The employee Mary is fixing it.

The employee Jane is fixing it.

The employee Tom is fixing it.

Notifying manager...

Asking for feedback from all supervisors...
Feedback by [Jack]
Feedback by [Jeff]


Notifying CEO...

Collecting decisions from managers...

Making final decisions...

Decision 0:
Evacuating members first...
Evacuating members first...
Evacuating members first...
The employee Nick has evacuated.
The employee John has evacuated.
The employee Mary has evacuated.
```

```
Evacuating members first...
The employee Nick has evacuated.
The employee John has evacuated.
The employee Mary has evacuated.
The employee Jane has evacuated.
The employee Tom has evacuated.
Employee Jack has evacuated
Evacuating members first...
The employee Rob has evacuated.
The employee Ed has evacuated.
The employee Rick has evacuated.
The employee Michael has evacuated.
Employee Jeff has evacuated
Employee Bob has evacuated
Evacuating members first...
Evacuating members first...
The employee Joe has evacuated.
The employee Frank has evacuated.
The employee Sam has evacuated.
The employee Greg has evacuated.
Employee Chuck has evacuated
Evacuating members first...
The employee Amy has evacuated.
The employee Will has evacuated.
The employee Nancy has evacuated.
The employee Adam has evacuated.
Employee Denise has evacuated
Employee Rachel has evacuated
Employee Steve has evacuated

Decision 1:
The city's environmental department is notified.


Process finished with exit code 0
```

Main:

```
import java.util.List;

import Aqua.Manager;
import Aqua.ProjectLeader;
import Aqua.Supervisor;
import Aqua.Worker;
import Green.Employee;
import Grey.Hazard;
import Aqua.CEO;

public class Main {
   public static void main(String[] args) {
        //first group of five employees (Boss = Jack -- Supervisor)
        Employee Nick = new Worker("Nick");
        Employee John = new Worker("John");
        Employee Mary = new Worker("Mary");
        Employee Jane = new Worker("Jane");
        Employee Tom = new Worker("Tom");

        List<Employee> underJack = List.of(Nick, John, Mary, Jane, Tom);

        //second group of four employees (Boss = Jeff -- Supervisor)
        Employee Rob = new Worker("Rob");
        Employee Ed = new Worker("Ed");
        Employee Rick = new Worker("Rick");
        Employee Michael = new Worker("Michael");

        List<Employee> underJeff = List.of(Rob, Ed, Rick, Michael);

        //third group of four employees (Boss = Chuck -- ProjectLeader)
        Employee Joe = new Worker("Joe");
        Employee Frank = new Worker("Frank");
        Employee Sam = new Worker("Sam");
        Employee Greg = new Worker("Greg");

        List<Employee> underChuck = List.of(Joe, Frank, Sam, Greg);

        //last group of four employees (Boss = Denise -- ProjectLeader)
        Employee Amy = new Worker("Amy");
        Employee Will = new Worker("Will");
        Employee Nancy = new Worker("Nancy");
        Employee Adam = new Worker("Adam");

        List<Employee> underDenise = List.of(Amy, Will, Nancy, Adam);


//-------------------------------------------------------------------------

        //First supervisor (Boss = Bob -- Manager)
        Employee Jack = new Supervisor("Jack", underJack);
```

```
        Nick.setDirectOverseer(Jack);
        John.setDirectOverseer(Jack);
        Mary.setDirectOverseer(Jack);
        Jane.setDirectOverseer(Jack);
        Tom.setDirectOverseer(Jack);

        //Second supervisor (Boss = Bob -- Manager)
        Employee Jeff = new Supervisor("Jeff", underJeff);

        Rob.setDirectOverseer(Jeff);
        Ed.setDirectOverseer(Jeff);
        Rick.setDirectOverseer(Jeff);
        Michael.setDirectOverseer(Jeff);


//-------------------------------------------------------------------------

        //First project leader (Boss = Rachel -- Manager)
        Employee Chuck = new ProjectLeader("Chuck", underChuck);

        Joe.setDirectOverseer(Chuck);
        Frank.setDirectOverseer(Chuck);
        Sam.setDirectOverseer(Chuck);
        Greg.setDirectOverseer(Chuck);

        //Second project leader (Boss = Rachel -- Manager)
        Employee Denise = new ProjectLeader("Denise", underDenise);

        Amy.setDirectOverseer(Denise);
        Will.setDirectOverseer(Denise);
        Nancy.setDirectOverseer(Denise);
        Adam.setDirectOverseer(Denise);


//-------------------------------------------------------------------------

        //First manager (Boss = Steve -- CEO)
        List<Employee> underBob = List.of(Jack, Jeff);

        Employee Bob = new Manager("Bob", underBob);

        Jack.setDirectOverseer(Bob);
        Jeff.setDirectOverseer(Bob);

        //Second manager (Boss = Steve -- CEO)
        List<Employee> underRachel = List.of(Chuck, Denise);

        Employee Rachel = new Manager("Rachel", underRachel);

        Chuck.setDirectOverseer(Rachel);
        Denise.setDirectOverseer(Rachel);
```

```
//--------------------------------------------------------------------------

        //Only CEO (Highest of the position)
        List<Employee> underSteve = List.of(Bob, Rachel);

        Employee Steve = new CEO("Steve", underSteve);

        Bob.setDirectOverseer(Steve);
        Rachel.setDirectOverseer(Steve);


//--------------------------------------------------------------------------

        //simulation

        //scenario
        System.out.println("A worker John observed a gas leak of a big tank and
triggered method to report it to his supervisor");
        //                                --type--        --description--
--hazard level--
        Hazard hazard = new Hazard("Gas leak", "Gas leak from a big tank", 10);
        John.seeDanger(null, hazard);
    }
}
```

CEO:

```
package Aqua;

import java.util.List;
import java.util.ArrayList;

import Green.Administrator;
import Green.Employee;
//import Green.Employee;
import Green.HReporter_IF;
import Grey.Decision;
import Grey.Hazard;

/*
* @brief CEO java class
*
* @note Extends to Administrator with relationship on lab
*/
public class CEO extends Administrator{

    /*
     * @brief Constructor
     * @param n : String
     * @param m : List<Employee>
     *
```

```
     * @note Since
     */
    public CEO(String n, List<Employee> m) {
        super(n, m);
    }

    public void implementDecision(List<Decision> ds){
        ds = sortByPriority(ds);

        for(int i = 0; i < 2; i++){
            System.out.println("\nDecision " + i + ":");
            ds.get(i).execute(this);
        }
    }

    private List<Decision> sortByPriority(List<Decision> ds){
        List<Decision> sortPriority = new ArrayList<>();

        while(!ds.isEmpty()){
            int index = 0;
            int priority = -1;
            for(int i = 0; i < ds.size(); i++){
                if(ds.get(i).getPriority() > priority){
                    priority = ds.get(i).getPriority();
                    index = i;
                }
            }
            sortPriority.add(ds.get(index));
            ds.remove(index);
        }
        return sortPriority;
    }

    @Override
    public void seeDanger(HReporter_IF r, Hazard h){
        List<Decision> ds = new ArrayList<>();

        System.out.println("Collecting decisions from managers...");
        for(HReporter_IF member : getMembers()){
            Manager manager = (Manager) member;
            ds.addAll(manager.suggestDecisions(h));
        }

        System.out.println("\nMaking final decisions...");
        implementDecision(ds);
    }

    public void report(){
        System.out.println("The city's environmental department is
notified.\n");
    }
}
```

Manager:

```
package Aqua;

import Green.Administrator;
import Green.DirectAdministrator;
import Green.Employee;
import Green.HReporter_IF;
import Grey.Decision;
import Grey.Evacuation;
import Grey.FileReport;
import Grey.Hazard;

import java.util.ArrayList;
import java.util.List;

public class Manager extends Administrator{
   //apparently making it a static boolean fixed the evacuation process, not
sure why -_-
   private static boolean report = false;

   public Manager(String n, List<Employee> members) {
       super(n, members);
   }

   @Override
   public void seeDanger(HReporter_IF r, Hazard h){
       boolean reportCEO = true;

       System.out.println("Asking for feedback from all supervisors...");
       for(HReporter_IF m : getMembers()){
           DirectAdministrator DA = (DirectAdministrator)m;
           if(!DA.getFeedback(h)){
               reportCEO = false;
           }
       }

       if(reportCEO && getDirectOverseer() != null){
           System.out.println("\n\nNotifying CEO...\n");
           getDirectOverseer().seeDanger(this, h);
       }

   }

   public List<Decision> suggestDecisions(Hazard h){
       List<Decision> suggestion = new ArrayList<>();
       int hazard_level = h.getHazardLevel();
       //boolean report = false; <-- epic fail

       if(!report){
           suggestion.add(new FileReport(this, 5, h));
           report = true;
       }
       else{
           if(hazard_level >= 10){
```

```
                suggestion.add(new Evacuation(this, 10, h));
            }else if(hazard_level >= 1){
                suggestion.add(new FileReport(this, 5, h));
            }
        }
        return suggestion;
    }
}
```

ProjectLeader:

```
package Aqua;
import java.util.List;
import Green.Employee;
import Green.DirectAdministrator;

public class ProjectLeader extends DirectAdministrator{

    /*
     * @brief Constructor
     */
    public ProjectLeader(String n, List<Employee> members){
        super(n, members);
    }


}
```

Supervisor:

```
package Aqua;

import java.util.List;

import Green.DirectAdministrator;
import Green.Employee;

public class Supervisor extends DirectAdministrator{

    public Supervisor(String n, List<Employee> members) {
        super(n, members);
    }

}
```

Worker:

```
package Aqua;

import Green.Employee;
import Green.HReporter_IF;
import Grey.Hazard;

public class Worker extends Employee{
```

```
    /*
     * @brief Constructor
     * @param n : String
     */
    public Worker(String n){
        super(n);
    }

    /*
     * @brief Triggers a report to notify the overseer of dangers occurred
     * @param r : HReporter_IF
     * @param h : Hazard
     *
     * @note following the format that was given on the lab (yellow sticky note)
     */
    @Override
    public void seeDanger(HReporter_IF r, Hazard h){
        if(r!=null){
            fixIt();
        }else {
            fixIt();
            System.out.println("Notifying direct overseer...\n");
            getDirectOverseer().seeDanger(this, h);
        }
    }

    /*
     * @brief Output statement that
     */
    @Override
    public void evacuate(){
        System.out.println("The employee " + getName() + " has evacuated.");
    }

    public void fixIt(){
        System.out.println("\nThe employee " + getName() + " is fixing it.");
    }
}
```

Administrator:

```
package Green;

import Grey.Hazard;
import java.util.List;

/*
* @brief Administrator java class
*
* @note Extending to Employee class of the clear arrow relationship
* @note Import List library due to multiplicity (*)
* @note Similar to Employee.java, had to make an abstract class since seeDanger
has no function body
```

```
*/
public abstract class Administrator extends Employee{
   public List<Employee> members;

   /*
    * @brief Constructor
    * @param n : String
    * @param m : List<Employee>
    *
    * @note call super as Employee is a super class
    * @note set members to ArrayList from List object
    */
   public Administrator(String n, List<Employee> m) {
       super(n);
       this.members = m;
   }

   //function will be called from Worker.java as indicated in the lab
   public abstract void seeDanger(HReporter_IF r, Hazard h);

   /*
    * @brief evacuate members from the employee class
    *
    * @note loop through each employee from the array list and evacuate
    * @note
    */
   public void evacuate(){
       System.out.println("Evacuating members first...");
       for(Employee e : members){
           e.evacuate();
       }
       System.out.println("Employee " + getName() + " has evacuated");
   }

   /*
    * @brief Getting members to be for the rest of the types of employees
within the Company XYZ
    * @return members
    */
   public List<Employee> getMembers(){
       return members;
   }
}
```

DirectAdministrator:

```
package Green;

import Grey.Hazard;
import java.util.List;
/*
* @brief DirectAdministrator java class
*/
public abstract class DirectAdministrator extends Administrator{
```

```
    //private int max_hazard;

    public DirectAdministrator(String n, List<Employee> members) {
            super(n, members);
            //max_hazard = 5;
        }

    @Override
    public void seeDanger(HReporter_IF r, Hazard h){

        System.out.println("Instructing all team members to fix it...");
        for(HReporter_IF m : getMembers()){
            if(m != r){
                m.seeDanger(this, h);
            }
        }

        if(getDirectOverseer() != null){
            System.out.println("\nNotifying manager...\n");
            getDirectOverseer().seeDanger(this, h);
        }
    }

    /*
     * @brief
     * @param hazard : Hazard
     */
    public boolean getFeedback(Hazard hazard){
        //boolean f = hazard.getHazardLevel() >= max_hazard;
        System.out.println("Feedback by [" + getName() + "]");
        return hazard.getHazardLevel() > 0;
    }

}
```

Employee:

```
package Green;
import Grey.Hazard;

/*
* @brief Employee java class
*
* @note include the employee name
* @note make each function public to have the visibility recognized from
HReporter_IF
* @note calling abstract would not require functions without function bodies
have syntax errors
*
* @note create a constructor to store the name value
*/
public abstract class Employee implements HReporter_IF{
    public String name;
```

```
    public Administrator overseer;

    /*
     * @brief Employee constructor
     * @param n : String
     */
    public Employee(String n){
        this.name = n;
    }

    //seeDanger func will be used in Worker.java
    public abstract void seeDanger(HReporter_IF r, Hazard h);

    //Evacuate func to be used in Administrator.java
    public abstract void evacuate();

    /*
     * @brief Set func for "-overseer" that is called from Administrator
     * @param r : HReporter_IF
     *
     * @note Used type casting for Adminstrator to assign overseer with passed
param of HReporter_IF
     */
    public void setDirectOverseer(HReporter_IF r){
        this.overseer = (Administrator) r;
    }

    /*
     * @brief Get func for overseer variable
     * @return overseer --> boss/administrator
     */
    public HReporter_IF getDirectOverseer(){
        return this.overseer;
    }

    public String getName(){
        return this.name;
    }

}
```

HReporter_IF:

```
package Green;

import Grey.Hazard;

/*
* @brief Public interface for HReporter
*
* @note import hazard java file to call the Hazard object in seeDanger function
*/
public interface HReporter_IF {
    public void seeDanger(HReporter_IF r, Hazard h);
```

```
   public void setDirectOverseer(HReporter_IF r);
   public HReporter_IF getDirectOverseer();
}
```

Decision:

```
package Grey;

import Aqua.Manager;
import Aqua.CEO;

public abstract class Decision {

   public Manager suggestedBy;
   public int priority;
   public Hazard hazard;

   public Decision(Manager s, int p, Hazard h){
       this.suggestedBy = s;
       this.priority = p;
       this.hazard = h;
   }

   public abstract void execute(CEO commander);

   public Manager getSuggestedBy(){
       return suggestedBy;
   }

   public Hazard getHazard(){
       return hazard;
   }

   public int getPriority() {
       return priority;
   }
}
```

Evacuation:

```
package Grey;

import Aqua.CEO;
import Aqua.Manager;

public class Evacuation extends Decision {

   public Evacuation(Manager s, int p, Hazard h){
       super(s, p, h);
   }

   @Override
   public void execute(CEO commander)
   {
```

```
        commander.evacuate();
    }
}
```

FileReport:

```
package Grey;

import Aqua.Manager;
import Aqua.CEO;

public class FileReport extends Decision {

    public FileReport(Manager s, int p, Hazard h){
        super(s, p, h);
    }

    @Override
    public void execute(CEO commander)
    {
        commander.report();
    }
}
```

Hazard:

```
package Grey;

public class Hazard {
    private String type;
    private String desp;
    private int hazard_level;

    /*
     * @brief Constructor
     * @param t : String
     * @param d : String
     * @param h : int
     */
    public Hazard(String t, String d, int h){
        this.type = t;
        this.desp = d;
        this.hazard_level = h;
    }
    /*
     * @brief Obtain the hazard level
     *
     * @note Used for DirectAdministrator getFeedback
     */
    public int getHazardLevel() {
        return hazard_level;
    }

    public String getType(){
```

```
        return type;
    }

    public String getDesp(){
        return desp;
    }
}
```