Problem 1

Provide definitions for the following terms.

- *Abstraction:* The set of concepts that some entity provides us to achieve a task.
- *Encapsulation:* Language-level hiding of implementation details.
- *Cohesion:* How closely the operations in a routine are related.
- *Coupling:* The strength of the connections between routines.
- *Also, how does each of these terms apply to the object-oriented notion of a class? Provide examples of both good and bad uses of these terms in the design of a class or a set of classes.*
  - *Abstraction:* The process of conceptualizing the functionality of a class at the design level to achieve a task or solve a particular problem.
    - Good use: The class Animal is an abstraction for a real animal. The attributes of this class could include foodLevel, location, etc. The methods of Animal define an Animal object's behaviors and could include eatFood(), moveLocation(), etc. This class can encompass more than one type of animal and thus can be an abstract class, with inheriting subclasses such as Cat and Dog.
    - Bad use: The class Animal attempts to represent all animals without enough abstraction levels, but its attributes and methods cannot differentiate between, for example, land and sea animals.
  - *Encapsulation:* A class can control the accessibility of their features, with visibility modes including public, protected, and private. Objects of the class don't expose their data members (attributes and methods) to the outside world if they are protected or private. Outsiders must use getter/setting methods to access them.
    - Good use: The SSN atribute of the class BankAccount is private and can only be accessed within the same class.
    - Bad use: The SSN attribute of the class BankAccount is public, allowing direct access from outside the class and unwantedly exposing the SNN of this account, risking security of the customer.
  - *Cohesion:* How closely methods in a class are related. Also referred to as clarity because the more that operations are related in a routine (or a class), the easier it is to understand things.
    - Good use: A method in a class performs just one operation and thus yields strong cohesion: The eat() method in the class Puppy only eats.

  - Bad use: A method in a class performs many operations and thus yields weak cohesion: The exist() method in the class Puppy sleeps, eats, plays, and meows all in one.

- *Coupling:* How connected/dependent classes are to one another. We want to create classes with loose coupling so they show small, direct, visible, and flexible relations to one another. Say we have a class Triangle and a class Area.
  - Good use: We get loose coupling when there is no dependency between the classes. If we change something in the Triangle class, we don't have to change the Area class.

```java
// Java example: Loose coupling
class Area {
    public static void main(String args[]) {
        Triangle t = new Triangle(5,5,5);
        System.out.println(t.getArea());
    }
}
final class Triangle {
    private int area;
    Triangle(int base, int height) {
        this.area = 1/2 * base * height;
    }
    public int getArea() {
        return area;
    }
}
```

  - Bad use: We get tight coupling when there is strong dependency between the classes. If there's any change in Triangle, that will alter the result in Area.

```java
// Java example: Tight coupling
class Area {
    public static void main(String args[]) {
        Triangle t = new Triangle(2,3);
        System.out.println(t.area);
    }
}
class Triangle {
    public int area;
    Triangle(int base, int height) {
        this.area = 1/2 * base * height;
    }
}
```

Problem 2

A company has asked us to design a payroll system that will pay employees for the work they perform each month. Using a level of abstraction, similar to that shown in Chapter 1 of the textbook and as shown on slide 6 of lecture 2, develop a design for this system using the functional decomposition approach. You can assume the existence of a database that contains all of the information you need on your employees. For your answer, first describe the functional decomposition approach, discuss what assumptions you are making concerning this problem, and then present your design.

Using the functional decomposition approach, we will break this problem into small steps, such as retrieving data from the database, processing the data, and then outputting the payroll for the employees. We will assume that the database is complete and accurate to our best knowledge. We will also assume that the company has provided complete, consistent, and verifiable requirements.

Then, our design is:

- Connect to database
- Locate and retrieve employee info (assumably containing: name, ID, direct deposit info, hours worked this month, employee type, etc.)
- We assume that the company has given us a list of specific employee types with different pay rates (such as manager, regular office worker, etc.)
- Looping through each employee in the list:
  - If the employee is of a specific type (e.g. manager):
    - Assuming the database doesn't already give us the total pay amount for each employee, we will calculate this using the employee-specific pay rates and their hours
    - Assuming all employees receive payment via direct deposit, we process payment and send it to their respective bank account
    - Print report of completion
  - Else if the employee of another type (i.e. regular office worker):
    - Calculate total pay
    - Process payment and send to their bank account
    - Print report
  - This if-else is done for each employee type

Problem 3

Now develop a design for the payroll system using the object-oriented approach, keeping in mind the points discussed on slides 21-23 of lecture 2 as well as the discussion in Chapter 1 of the textbook. Identify the classes you would include in your design and their responsibilities. (As before, you can assume the existence of a database and that you'll be able to create objects based on the information stored in that database.) Then, identify what objects you would instantiate and in what order and how they would work together to fulfill the responsibilities associated with the payroll system.

For simplicity, let's assume we just have two types of employees: managers and regular office worker. The following table lists the classes in the design and their responsibilities.

Class	**Responsibilities (Methods)**
EmployeeDatabase	getCollection – Get sorted (alphabetically by last name) collection of employee data
Employee (abstract class)	getData – Return data for Employee (e.g. name, ID, employee type, etc.)
Manager (derived from Employee)	processData – Read and process manager employee data computePay – Calculate pay according to hours and manager pay rate printReport – Print report of payroll
OfficeWorker (derived from Employee)	processData – Read and process regular office worker employee data computePay – Calculate pay according to hours and regular office worker pay rate printReport – Print report of payroll
Collection	processData – Tell all contained employees to read and process data computePay – Tell all contained employees to compute pay printReport – Tell all contained employees to print report

Main program with objects to instantiate in order:

1. Create an instance of the database object
2. Ask the database object to find the sorted collection of employee data we are interested in and to instantiate a collection object containing all the employees
3. The collection asks the employees it contains to process their data
4. The collection asks the employees it contains to compute their pay
5. The collection asks the employees it contains to print their reports

With the object-oriented approach, we can add a new type of employees by creating a new derivation of Employee and in that derivation, implement the methods (processData, computePay, printReport) appropriate for that employee type.