San José State University
Computer Science Department
CS151, Object Oriented Design and Programming, 07, Spring 2021

Homework #3

Objective:
This homework's objective is to review and understand the unit on object oriented design principles and various ways to utilize and implement them in Java.

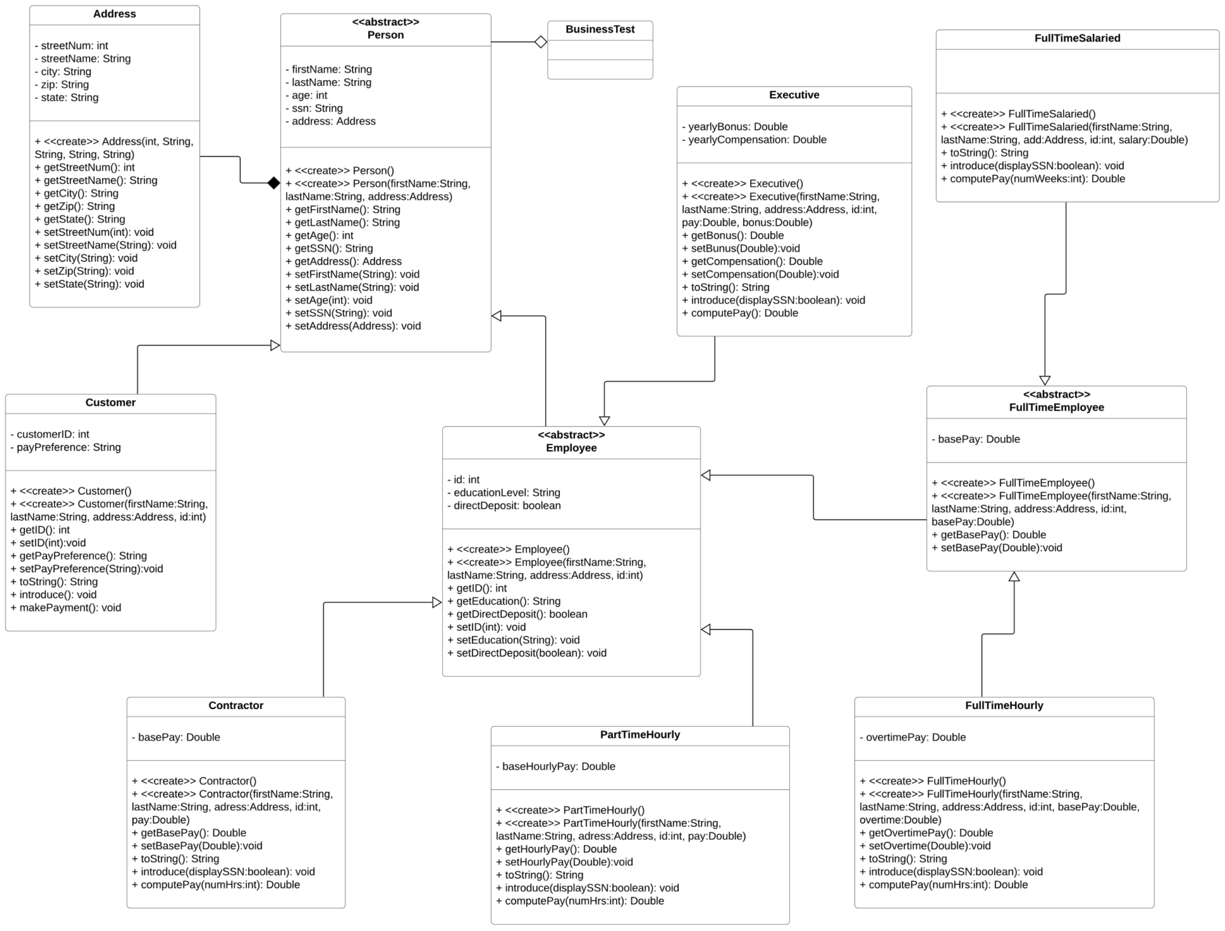
Details:
Exercise 1:
Food for thought:
When you start working on a new application/project you usually start with gathering the requirements for what types of things your application should do and what types of problems it should solve. Usually these requirements will come from domain experts and/or marketing research at your company. As a rule, these requirements will not tell you how to design your application but will describe the end results that your application is required to produce. However, these requirements will influence how you decide to design your internal application architecture. Depending on the application, the company structure, and many other factors, these design decisions are made either by you alone or by a team of people. In real life, your initial design will probably be updated and modified in minor and/or major ways throughout the lifetime of your application. Some of the reasons these changes happen are: feedback from the users, changes in the domain to which the application applies, changes to the types of problems the application is trying to solve, and many more. Major changes are usually costly and time consuming to implement. You should always make effort to design your application in a way that minimizes these future changes and try to anticipate as many revisions/additions to the requirements. Design your application from the beginning in a way that makes it easier to make small changes in the future, without overhauling the whole system.

Let's say, you have an application that models a business and how it runs. This business consists of customers and employees. There are different types of employees. For example, hourly and salaried employees, full time and part time employees. There are also contractors and executives. How would you design such an application? Some of the things to think about is that both customers and employees are just different types of persons operating within this business infrastructure. This could be represented by introducing a class or an interface called Person and have Customer and employee classes inherit from that entity. Collectively, all employees could be modeled by a single class or an interface called Employee, with individual employee types being modeled by classes that inherit from a common parent Employee entity. Do you want to allow instantiating instances of a generic employees? If not, make that parent entity an abstract class or an interface. Remember why we make these design decisions? We want to avoid repeating the same attribute fields in multiple classes. We also want to avoid implementing the same methods in multiple classes. If some information in multiple classes can be combined then always provide additional layers of inheritance and abstraction by moving those attributes and methods to the parent entity.

Why do we care about not duplicating code and information? One of the major reasons is maintainability, although others come in play too: scalability, clarity, convenience, ease of introducing new developers to your code and many more. Avoiding code duplication is one of the ways to make future changes to your applications easier and less costly. Let's say, you have to make a change to a method logic. If the same method implementation shows up in multiple classes then you have to find every place in your code where you need to make the change and copy-paste your code multiple times. Note, that we are not talking about method overriding here, which is a commonly used form of polymorphism and has many valid uses in the world of object oriented design. What we want to avoid is full method logic duplication. Similar thinking goes for attributes. In the field of information theory the rule is to avoid data duplication as much as possible. If there are three variables (A, B, and C) and variable C can be computed using A and B then it is advised to not store variable C, but instead compute that information from stored A and B every time. Over time, different pieces of information might become out of sync and if we rely on retrieved value of C but A and B have changed without C being updated, then we risk working with out-of-date information. This can be absolutely critical depending on the type of application you are working on. Therefore, when talking about class attributes do your best to design classes in a way that different attributes are independent and do not carry the same or partially the same information as other attributes in the same class. And as the case with code duplication, avoid having the same attributes in multiple classes. Introduce levels of abstraction and inheritance where necessary. In case of our business application example, every person has a name and an address (for simplicity of this example we ignore the reality that some people actually do not have an address or social security numbers, but something to consider if you are working on a real life application). Having the name and the address, as attributes, in a Customer class or employee classes violates the idea of not duplicating information in multiple classes. The better solution is to create a Person class and add the name and the address attributes to it, which Customer and employee classes inherit from it. There are some additional things you might consider when designing your class hierarchy. Do you want to group all full time employees under a single parent? Do you want to group hourly paid vs. salary paid employees under a single parent? Any other levels of abstraction you want to add? Which classes do you want to make abstract or concrete? Which entities are better being declared as interfaces? You need to consider the fact that Java does not allow multiple class inheritance, so this decision might be guided by this fact.

The final note will be about not overcomplicating things. While you want to create an appropriate hierarchy of abstractions, you do not want to make your application design so complex that it makes it difficult to understand and maintain. Keep things simple enough to be practical. Below you are asked to implement an application that models a business. I ask you to implement a specific design that follows the provided UML case diagram. Remember that this is one particular manifestation of a design for such an application. Many equally valid designs can be devised for this application and it is very possible that, given a chance, you would make slightly different decisions. This exercise is meant to provoke your thoughts about how you would go about designing an application and what types of questions you would ask.

Exercise instructions:
Design and implement the below class hierarchy representing individuals involved in a business.


Address
- streetNum: int
- streetName: String
- city: String
- zip: String
- state: String
+ <<create>> Address(int, String, String, String, String)
+ getStreetNum(): int
+ getStreetName(): String
+ getCity(): String
+ getZip(): String
+ getState(): String
+ setStreetNum(int): void
+ setStreetName(String): void
+ setCity(String): void
+ setZip(String): void
+ setState(String): void
<<abstract>> Person
- firstName: String
- lastName: String
- age: int
- ssn: String
- address: Address
+ <<create>> Person()
+ <<create>> Person(firstName:String, lastName:String, address:Address)
+ getFirstName(): String
+ getLastName(): String
+ getAge(): int
+ getSSN(): String
+ getAddress(): Address
+ setFirstName(String): void
+ setLastName(String): void
+ setAge(int): void
+ setSSN(String): void
+ setAddress(Address): void
BusinessTest
FullTimeSalaried
+ <<create>> FullTimeSalaried()
+ <<create>> FullTimeSalaried(firstName:String, lastName:String, add:Address, id:int, salary:Double)
+ toString(): String
+ introduce(displaySSN:boolean): void
+ computePay(numWeeks:int): Double
Executive
- yearlyBonus: Double
- yearlyCompensation: Double
+ <<create>> Executive()
+ <<create>> Executive(firstName:String, lastName:String, address:Address, id:int, pay:Double, bonus:Double)
+ getBonus(): Double
+ setBunus(Double):void
+ getCompensation(): Double
+ setCompensation(Double):void
+ toString(): String
+ introduce(displaySSN:boolean): void
+ computePay(): Double
Customer
- customerID: int
- payPreference: String
+ <<create>> Customer()
+ <<create>> Customer(firstName:String, lastName:String, address:Address, id:int)
+ getID(): int
+ setID(int):void
+ getPayPreference(): String
+ setPayPreference(String):void
+ toString(): String
+ introduce(): void
+ makePayment(): void
<<abstract>> Employee
- id: int
- educationLevel: String
- directDeposit: boolean
+ <<create>> Employee()
+ <<create>> Employee(firstName:String, lastName:String, address:Address, id:int)
+ getID(): int
+ getEducation(): String
+ getDirectDeposit(): boolean
+ setID(int): void
+ setEducation(String): void
+ setDirectDeposit(boolean): void
<<abstract>> FullTimeEmployee
- basePay: Double
+ <<create>> FullTimeEmployee()
+ <<create>> FullTimeEmployee(firstName:String, lastName:String, address:Address, id:int, basePay:Double)
+ getBasePay(): Double
+ setBasePay(Double):void
Contractor
- basePay: Double
+ <<create>> Contractor()
+ <<create>> Contractor(firstName:String, lastName:String, adress:Address, id:int, pay:Double)
+ getBasePay(): Double
+ setBasePay(Double):void
+ toString(): String
+ introduce(displaySSN:boolean): void
+ computePay(numHrs:int): Double
PartTimeHourly
- baseHourlyPay: Double
+ <<create>> PartTimeHourly()
+ <<create>> PartTimeHourly(firstName:String, lastName:String, adress:Address, id:int, pay:Double)
+ getHourlyPay(): Double
+ setHourlyPay(Double):void
+ toString(): String
+ introduce(displaySSN:boolean): void
+ computePay(numHrs:int): Double
FullTimeHourly
- overtimePay: Double
+ <<create>> FullTimeHourly()
+ <<create>> FullTimeHourly(firstName:String, lastName:String, address:Address, id:int, basePay:Double, overtime:Double)
+ getOvertimePay(): Double
+ setOvertime(Double):void
+ toString(): String
+ introduce(displaySSN:boolean): void
+ computePay(numHrs:int): Double

A business might have executives, full time salaried employees, full time hourly employees, part time hourly employees, hourly paid contractors, as well as customers/clients. Each individual might have such attributes as first name, last name, age, social security number, address, level of education, payment method preference, direct deposit or not, id number (could be customer number or employee number), hourly pay, overtime pay, yearly salary, yearly bonus. Not all of these attributes are relevant to every type of person, while others are common to all individuals. For example, payment method preference is only relevant to customers.

The address should not be a simple *String* object. It should be a separate class with individual fields defined for street number, street name, city name, zip code, and state. For this exercise, we will assume that no person has an apartment number or a second address line.

Note that some of the described above attributes, in both Address class and other classes, would normally be enums, not String objects, e.g. state, payment method preference, etc. Because we have not covered enums in this class yet I ask you to implement them as String objects. However, if you are comfortable with working with enums you are free to implement these attributes as such.

Each class needs to have the constructors described in the class diagram. For example, the constructor for an object of type Person accepts first and last name and address as input arguments. For some classes additional information would need to be provided to that constructor, e.g. base pay amount (see the diagram for each class). In addition, each class also needs to either have getters and setters defined and implemented or inherit them for the parent attributes.

Implement *toString()* method in each of the leaf classes: executive, full time salaried employee, full time hourly employee, part time hourly employee, hourly paid contractor, and customer. This method should return a *String* object that contains text information about all the attribute values, including the type of person, for that class instance. Feel free to use inheritance and implement toString() method in Person class and invoke or override it as needed in child classes. Also implement method *introduce()* that displays the information returned by *toString()* method to command line. Implement a variation of method *introduce()* to flag if social security number should be displayed or not. For objects of Customer type *introduce()* method should just never display SSN information.

Implement *computePay()* method in each of the leaf classes, except Customer class. For an executive, this method will not accept any input arguments. For a full time salaried employee, this method will accept the number of weeks as an input argument. For a part time hourly employee, a full time hourly employee, or an hourly paid contractor, this method will accept the number of hours as an input argument. For all of these individuals, this method will return a floating point value indicating the value of the pay for the specified input arguments. Remember that the logic of computing the pay is different for the different individuals. For an executive, the pay is simply their yearly compensation plus the yearly bonus. For a full time salaried employee, the pay is based on their yearly compensation adjusted to the number of weeks specified by the input argument. For a full time hourly employee, the pay is based on the hourly pay and the number of hours specified by the input argument. Make sure to account for possible overtime. For a part time hourly employee, the pay is based on the hourly pay and the number of hours specified by the input argument. Part time hourly employees are not allowed to work more than 40 hours a week, so implement a check for the proper number of hours and if they exceed 40 hours, return -1. For an hourly paid contractor, the pay is calculated by simply multiplying the number of hours they worked by the base pay.

Implement *makePayment()* method only available to customers. This method should display the preferred method of payment for the customer to command line screen.

Define and implement class **BusinessTest**. This class should implement *main()* method. In the body of the *main()* method you should create at least 2 instances of each of the leaf classes: executive, full time salaried employee, full time hourly employee, part time hourly employee, hourly paid contractor, and customer. It is up to you to choose which values of the attributes for each of the instances to use. For each instance, make a call to *introduce()* method and make a call to *computePay()* method for business associates and *makePayment()* for customers. Again, it is up to you to choose the input arguments into *computePay()* method where they are required. For visual presentation make sure to include an empty line between each employee instance output. Save this class and its definition into a file named **BusinessTest.java**.

Submission:

In your class repo create a directory called "Assignment3" and add all the files created for this homework assignment to that directory. The following files are expected to be in this directory: BusinessTest.java, Address.java, Person.java, Employee.java, Customer.java, Contractor.java, PartTimeHourly.java, Executive.java, FullTimeEmployee.java, FullTimeSalaried.java, FullTimeHourly.java.

Make sure to email this submission by 11:59pm on the due date listed in Canvas. Email your assignment submission to me at both Yulia.Newton@sjsu.edu and yulia.newton@gmail.com, as well as the grader at akshay.kajale@sjsu.edu. The subject of the email should say "CS151 Assignment 3". Add your name as it appears on the class roster and the URL to your git repo in the body of an email.

In the prior semesters the grader and I had various git permission issues with some homework submissions, so we want you to also attach all the files for this homework as an attached single compressed file.

Grading:

Your code must compile and execute successfully in order to get full credit for this assignment. I will compile and execute BusinessTest.java file listed in that exercise description.

- Program with no compile errors (5 pts)
- Program executes (5 pts)
- Program outputs what is required by the exercise (5 pts)

A total of 15 points are possible for this homework assignment.