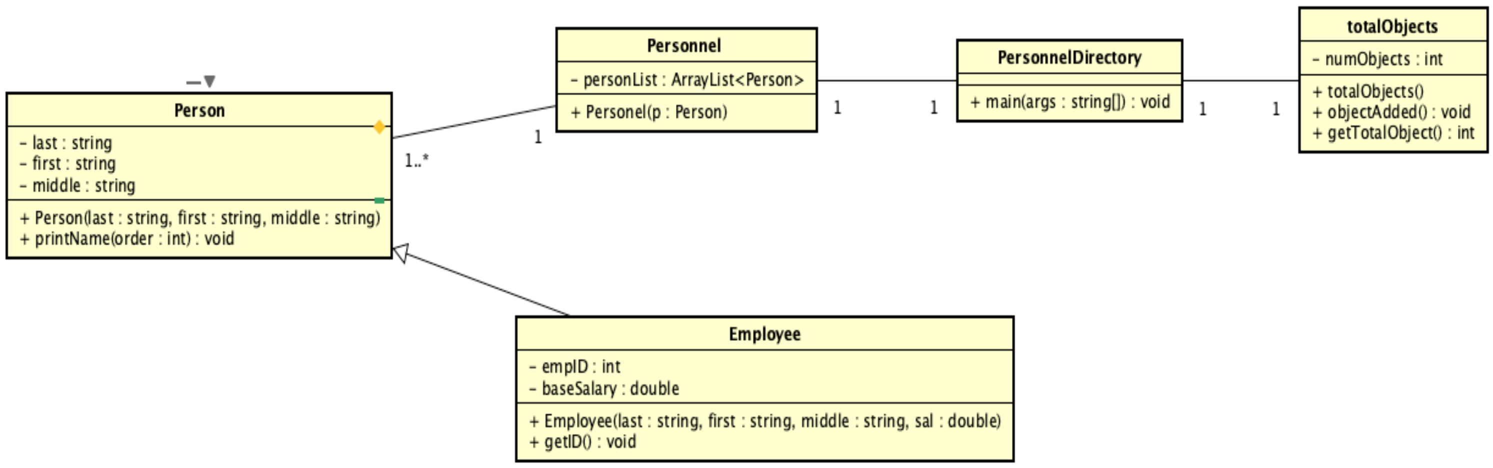
CSE 598 ASAD Fall A 2024
Miguel Castillo

Directory Management System Project

Phase I Part 1

Use Astah to draw a class diagram diagram. Use proper UML notation. Take a clear screenshot of your completed diagram and paste it on this page.



Phase I Part 2

Identify the places in the code where there are object-oriented concept violations, content coupling, common coupling, control coupling, and stamp coupling situations. On this page, paste the code segments that correspond to each situation and explain how you would fix object-oriented concept violations, common coupling, control coupling, and content coupling issues. You may add pages if necessary.

Object-Oriented Concept Violations

```
public class Person {
        public String last;
        public String first;
        public String middle;
.
.
.
}
```

Problem: Violation of principle of encapsulation: In the Person class, the attributes first, last, and middle are public. Currently, they are accessible directly by other class.

Solution: attributes of first, last, and middle, should be written as "private", and a provide a public getter and setter methods to access and modify them. Java code to fix issue is below:

```
public String getFirstName() { return first; } public void setFirstName(String first) { this.first = first; }
public String getLastName() { return last; } public void setLastName(String last) { this.last = last; }
public String getMiddleName() { return middle; } public void setMiddleName(String middle) {
this.middle = middle; }
```

Single Responsibility Violation

```
public class PersonnelDirectory
{
  public static void main(String[] args)
  {
          Personnel per = new Personnel();
                        totalObjects total = new totalObjects();
                        Scanner scan = new Scanner(System.in);
                        String firstN, lastN, middleN;
                        int empID;
```

```
                                double salary;
             int choice = -1;

      do{


          System.out.println("Welcome to the Personnel Directory Management System");
          System.out.println("=======================================================");
.
.
.
.
switch(choice){
.
.
   per.addPersonnel(e1);
   total.objectAdded();
.
.
} while(true);

  }
}
```

<u>Problem:</u> The main() method within the PersonnelDirectory class, is currently managing both personnel and handling user input and output. This class should only be concern with one responsibility and not two.

<u>Solution:</u> Separate the user interface logic and personnel management logic into two. Keep the current PersonnelDirectory class to only handle user interface logic.
Also make a new “PersonnelManager” class for the core business logic related to personnel for adding, finding, and printing names. Fix:

```
public class PersonnelManager {
private Personnel personnel = new Personnel();
//Adding, finding and printing names logic…
.
.
}

public class PersonnelDirectory { public static void main(String[] args) {
```

```
PersonnelManager manager = new PersonnelManager();
.
.
//User interface logic…
.
}
```

Open/Closed Violation

```
public class Person {
.
.
.

public void printName(int order)
        {

          if(order == 0)
          {
            System.out.println(first + "  " + middle + "  " + last);

          }else if(order == 1)
             {

             System.out.println(first + " ," + last+ " " + middle);

             }
             else if(order == 2)
                              {

                              System.out.println(last + " ," + first + " " + middle);

             }
        }
}
```

Problem: printName(int order) method in Person violates the Open/Closed Principle. This means that classes should be open for extension but closed for modifications. Currently, the “int order” is being used to control the format and limits flexibility if new requirements/functionality needs to be implemented.

Solution: Use polymorphism to create separate methods or classes of different print formats. This will allow future extension without modifications. In other words, this will keep the name printing logic generic thus making the class loosely coupled. Lastly, implement a different class for each format. Solution code:

```
public interface NamePrinter {
   void printName(Person person);
}

public class FirstMiddleLastPrinter implements NamePrinter {
   @Override
   public void printName(Person person) {
      System.out.println(person.getFirstName() + " " + person.getMiddleName() + " " +
person.getLastName());
   }
}

public class FirstLastMiddlePrinter implements NamePrinter {
   @Override
   public void printName(Person person) {
      System.out.println(person.getFirstName() + " " + person.getLastName() + " " +
person.getMiddleName());
   }
}

public class LastFirstMiddlePrinter implements NamePrinter {
   @Override
   public void printName(Person person) {
      System.out.println(person.getLastName() + ", " + person.getFirstName() + " " +
person.getMiddleName());
   }
}
```

Content Coupling

```
if( per.personList.get(i).first.equals(firstN) && per.personList.get(i).last.equals(lastN))
{
.
.
.
}
```

Problem: In PersonnelDirectory, the code accesses the attributes of Person directly as per.personList.get(i).first .
Solution: Make first, last, and middle in Person private and create public getter methods like getFirstName(), getLastName(), and getMiddleName() . Solution code:

```
public String getFirstName() { return first; }
public void setFirstName(String first) { this.first = first; }
public String getLastName() { return last; }
 public void setLastName(String last) { this.last = last; }
public String getMiddleName() { return middle; }
public void setMiddleName(String middle) { this.middle = middle; }
```

Control Coupling

```
public void printName(int order)
        {

           if(order == 0)
           {
             System.out.println(first + "  " + middle + "  " + last);

           }else if(order == 1)
              {

              System.out.println(first + " ," + last+ " " + middle);

              }
              else if(order == 2)
                              {

                              System.out.println(last + " ," + first + " " + middle);

              }
        }
```

<u>Problem:</u> The order parameter dictates which format to print the name in, controlling the behavior of printName.

<u>Solution:</u> To reduce control coupling, we can replace printName(int order) with individual methods such as printNameFirstMiddleLast(), printNameFirstLastMiddle(), and printNameLastFirstMiddle(). This helps with having a single responsibility and isn't controlled by a parameter. Solution code:

```
public void printNameFirstMiddleLast() { ... }
public void printNameFirstLastMiddle() { ... }
public void printNameLastFirstMiddle() { ... }
```

Common Coupling

```
public class totalObjects
{
  private static int numObjects = 0;
.
.
}
```

<u>Problem:</u> This type of issue occurs when multiple classes share a global variable which leads to dependencies across classes. In this case, the static member "numObjects" is associated with the class itself and not with a particular instance of this class, meaning it would be global throughout all instances. Although only one instance of numObjects is created and used, we can say that this is an issue if another instance were created.

<u>Solution:</u> I think we can fix this choosing one of two solutions. One could be to get rid of numObjects and to just directly get the number of total objects within Personnel class with personLIst.size(). The second solution is to make encapsulate totalObjects, make it private to one class and have it accessed through PersonnelDirectory via a setter and getter methods. This would make totalObjects not global. Solution code:

```
public class Personnel {
   private static int numObjects = 0; // Private and static to Personnel

   // Method to increment the count
   public void addPersonnel(Person person) {
      personList.add(person);
      incrementObjectCount(); // Only accessible within Personnel
   }
```

```
    // Private method to manage numObjects within this class
    private static void incrementObjectCount() {
        numObjects++;
    }

    // Public method to retrieve the count outside this class
    public static int getTotalObjects() {
        return numObjects;
    }
}
```

Also here is another common coupling issue:

Case 2:

```
.
.
if(found)  {
System.out.println("Found");
per.personList.get(loc).printName(0);
}else
  {
System.out.println("not found");
 Person p1  = new Person(lastN, firstN, " ");
per.personList.add(p1);
total.objectAdded();
}
```

Problem: When searching for a name and if its not found, the code creates a new person. Since the array list is shared globally, can create unexpected behavior across the system, especially if other parts of the code are unaware that a new person might be added automatically.
This can lead to inconsistent states or unintended side effects, as different parts of the code are implicitly dependent on personList's state.

Solution: I would say to separate the searching and creating the new person separately. Offer the user the option to add a new person manually through a separate action, such as case 3 for adding a new entry. Solution:

```
else {
        System.out.println("Person not found. Would you like to add them? (yes/no)");
        String response = scan.nextLine();
```

```
    if (response.equalsIgnoreCase("yes")) {
        System.out.println("Enter personnel type (Employee, Executive, etc.): ");
        String type = scan.nextLine();

        // Create the personnel using the factory
        Printable newPerson = PersonnelFactory.createPersonnel(type, lastN, firstN, " ");
        if (newPerson != null) {
            manager.addPersonnel((Person) newPerson);
            total.objectAdded();
            System.out.println("New " + type + " added to directory.");
        } else {
            System.out.println("Invalid personnel type.");
        }
    }
```

Stamp Coupling

```
Public class PersonnelDirectory{
for(int i =0; i <per.personList.size(); i++) {
        if( per.personList.get(i).first.equals(firstN) && per.personList.get(i).last.equals(lastN))
        {
         .
         .
         .
          Person p1  = new Person(lastN, firstN, " ");
          per.personList.add(p1);
          total.objectAdded();
         }
}
```

Problem: Stamping coupling occurs when an object is passed or created with more data than it actually needs. In this case, the Person contructor is given an empty string as an argument to satisfy the “middle” parameter for the print line within printName() method.

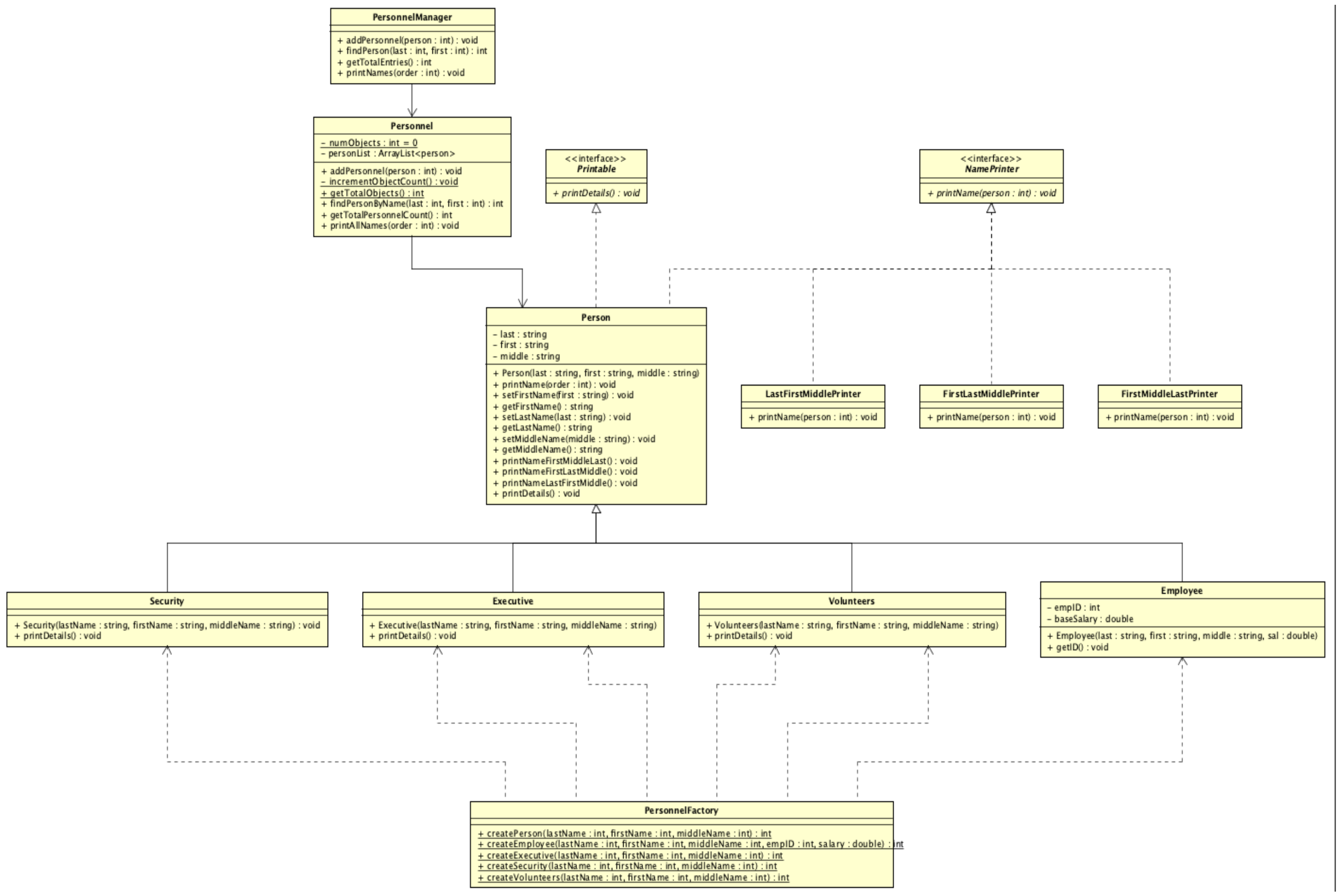
Solution: We can add an overloaded constructor in Person class that accepts only “first” and “last” names, thus eliminating the empty string placeholder for “middle”. Also set this.middle = null; within the overloaded constructor. Solution code:

```
public class Person {
.
   // Constructor with first, middle, and last
   public Person(String last, String first, String middle) {
       this.last = last;
       this.first = first;
       this.middle = middle;
   }

   // Constructor with only first and last
   public Person(String last, String first) {
       this.last = last;
       this.first = first;
       this.middle = null;
   }}
```

Phase II Part 2

After you have incorporated the PersonnelFactory, use Ashta to draw a UML class diagram of the Personnel Directory. Use proper UML notation. When you have completed your diagram, take a clear screenshot of your diagram and paste it on this page.


PersonnelManager
+ addPersonnel(person : int) : void
+ findPerson(last : int, first : int) : int
+ getTotalEntries() : int
+ printNames(order : int) : void
Personnel
– numObjects : int = 0
– personList : ArrayList<person>
+ addPersonnel(person : int) : void
– incrementObjectCount() : void
+ getTotalObjects() : int
+ findPersonByName(last : int, first : int) : int
+ getTotalPersonnelCount() : int
+ printAllNames(order : int) : void
<<interface>>
Printable
+ printDetails() : void
<<interface>>
NamePrinter
+ printName(person : int) : void
Person
– last : string
– first : string
– middle : string
+ Person(last : string, first : string, middle : string)
+ printName(order : int) : void
+ setFirstName(first : string) : void
+ getFirstName() : string
+ setLastName(last : string) : void
+ getLastName() : string
+ setMiddleName(middle : string) : void
+ getMiddleName() : string
+ printNameFirstMiddleLast() : void
+ printNameFirstLastMiddle() : void
+ printNameLastFirstMiddle() : void
+ printDetails() : void
LastFirstMiddlePrinter
+ printName(person : int) : void
FirstLastMiddlePrinter
+ printName(person : int) : void
FirstMiddleLastPrinter
+ printName(person : int) : void
Security
+ Security(lastName : string, firstName : string, middleName : string) : void
+ printDetails() : void
Executive
+ Executive(lastName : string, firstName : string, middleName : string)
+ printDetails() : void
Volunteers
+ Volunteers(lastName : string, firstName : string, middleName : string)
+ printDetails() : void
Employee
– empID : int
– baseSalary : double
+ Employee(last : string, first : string, middle : string, sal : double)
+ getID() : void
PersonnelFactory
+ createPerson(lastName : int, firstName : int, middleName : int) : int
+ createEmployee(lastName : int, firstName : int, middleName : int, empID : int, salary : double) : int
+ createExecutive(lastName : int, firstName : int, middleName : int) : int
+ createSecurity(lastName : int, firstName : int, middleName : int) : int
+ createVolunteers(lastName : int, firstName : int, middleName : int) : int