Software Engineering and Management
DIT043 - Object-oriented programming

Assignment 3 - Template Report

Name: Joel Mattsson

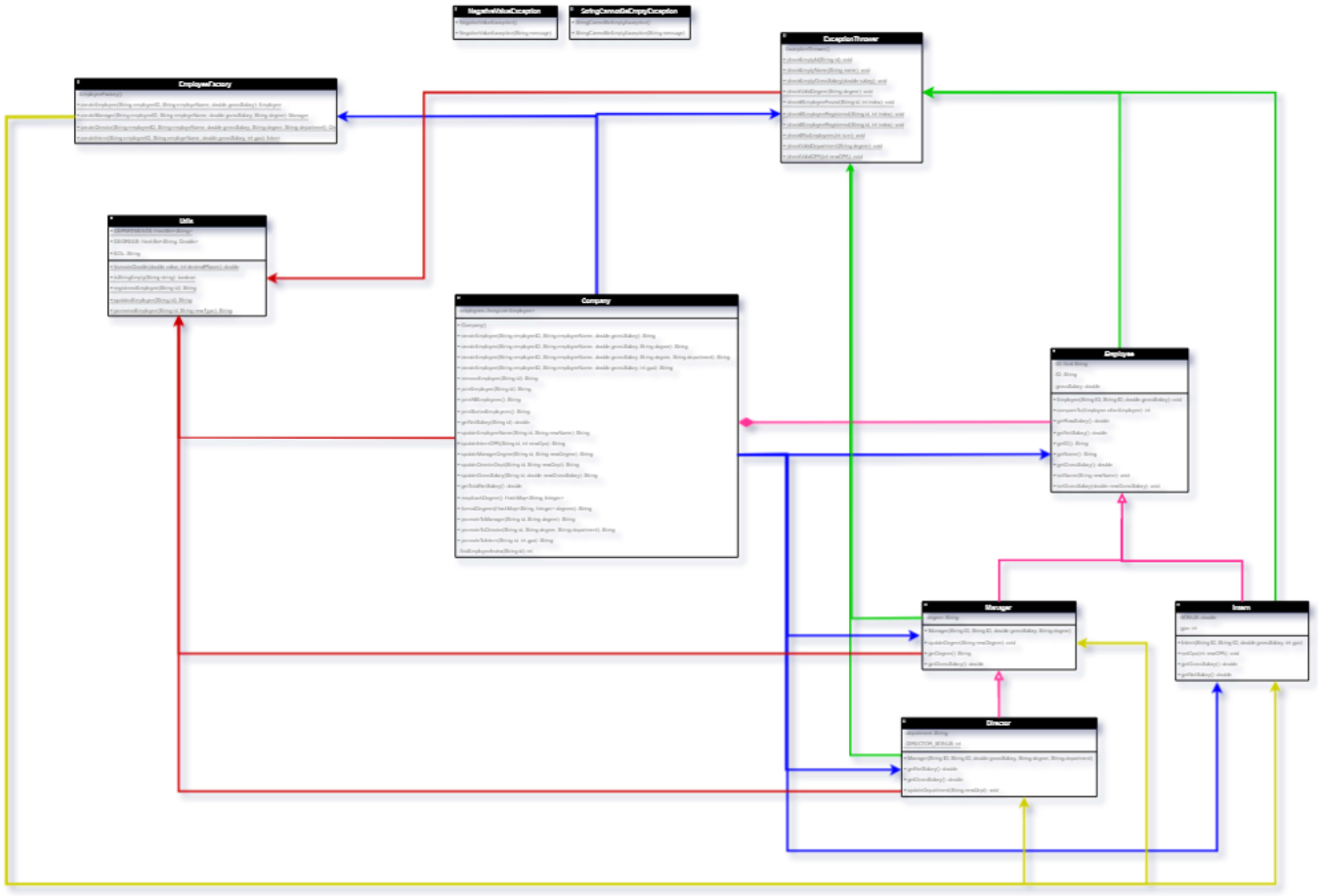
E-mail: gusmatjoao@student.gu.se

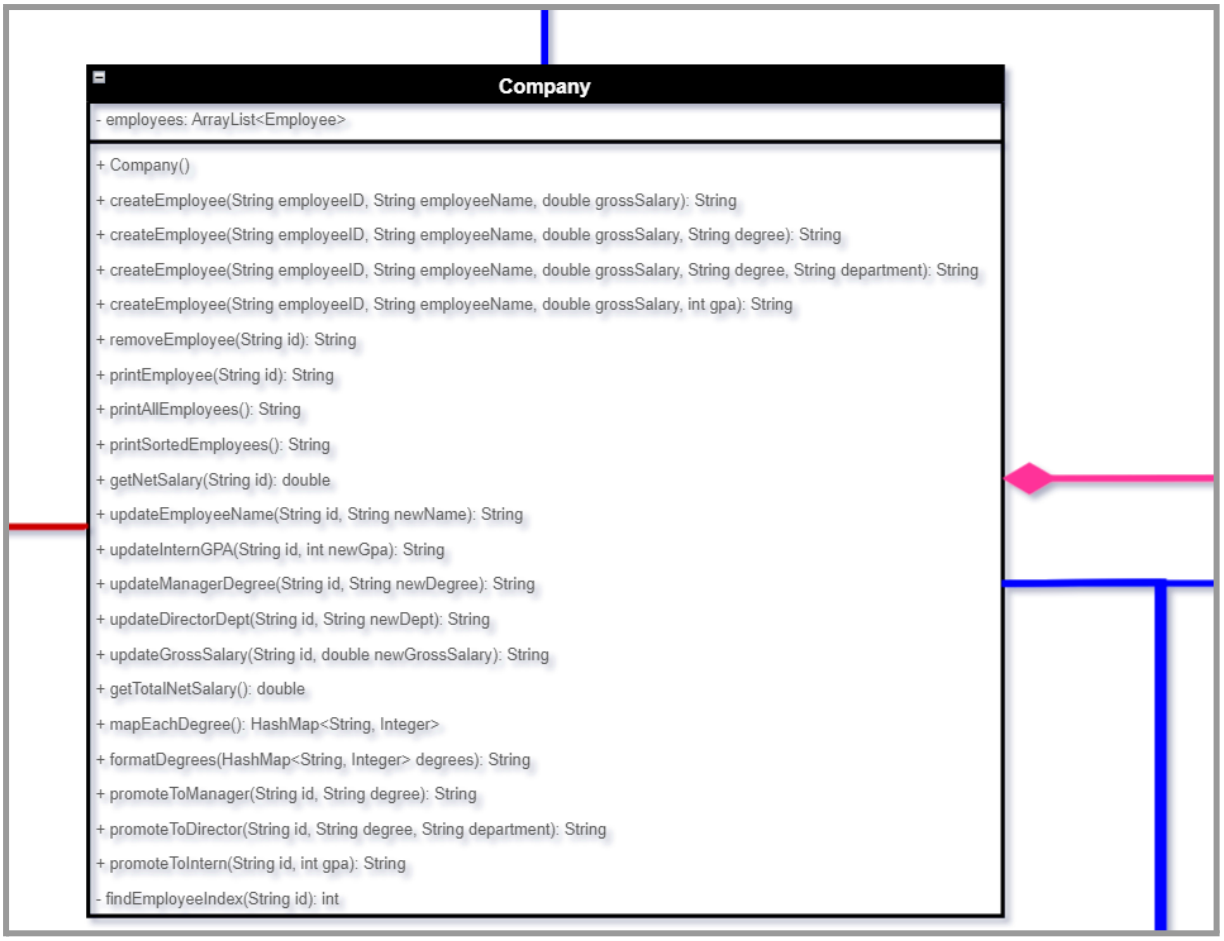
1. Class Diagram

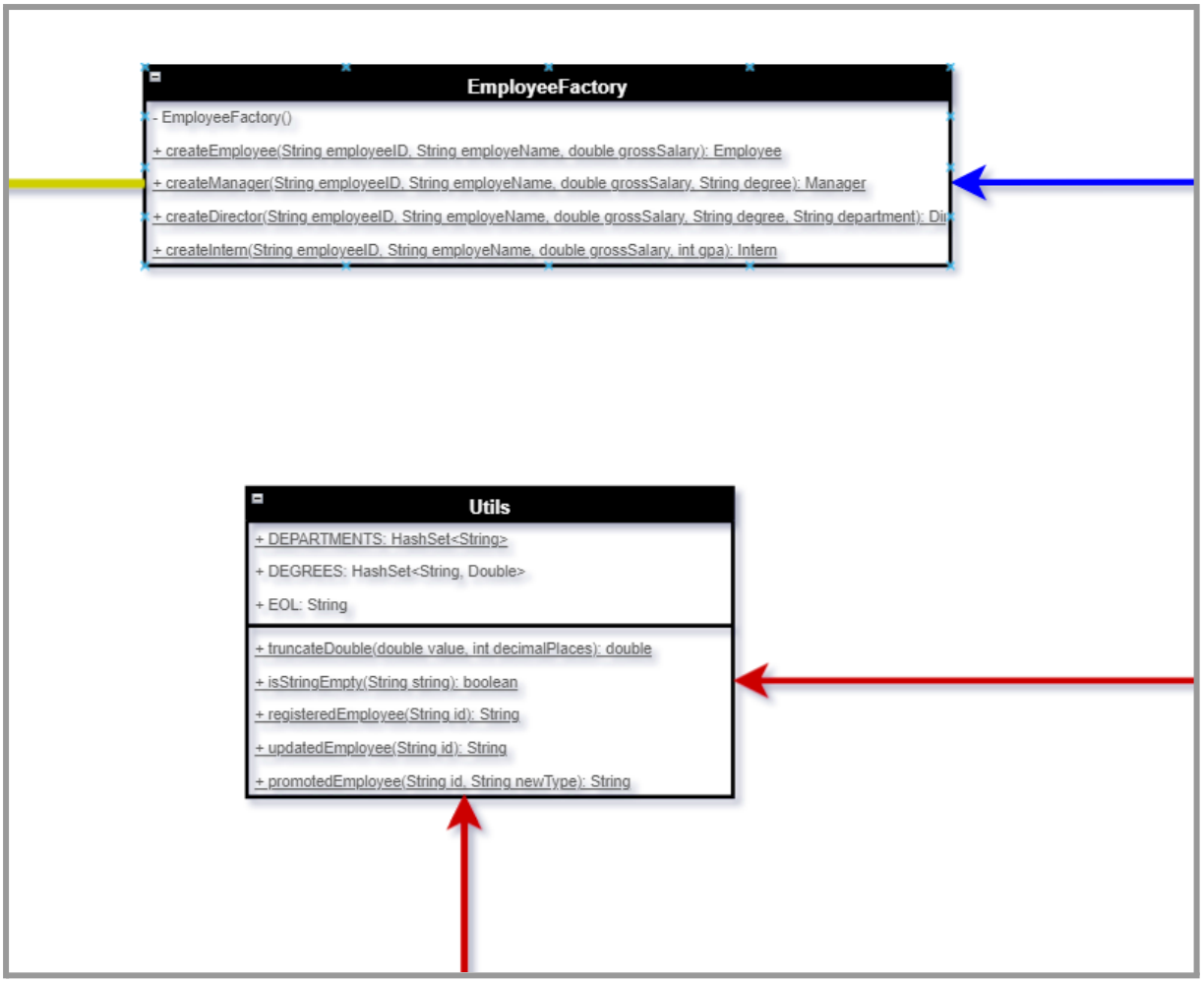
- Underline: static, ALL_CAPS: final, Made in draw.io
- Includes inheritance, composition and association (not aggregation)

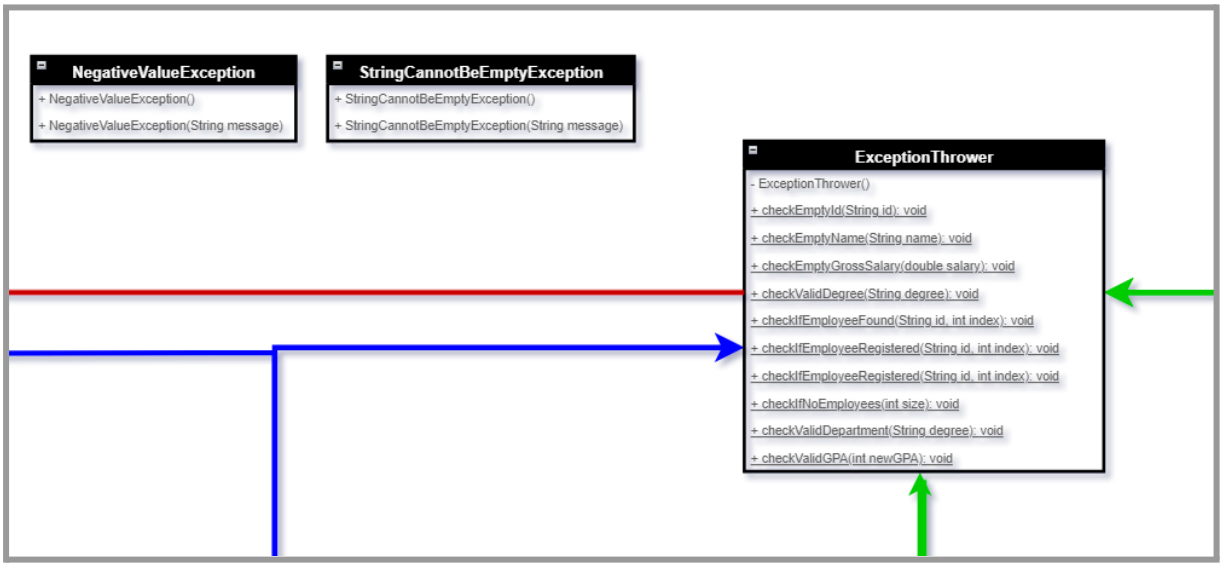
Organization:

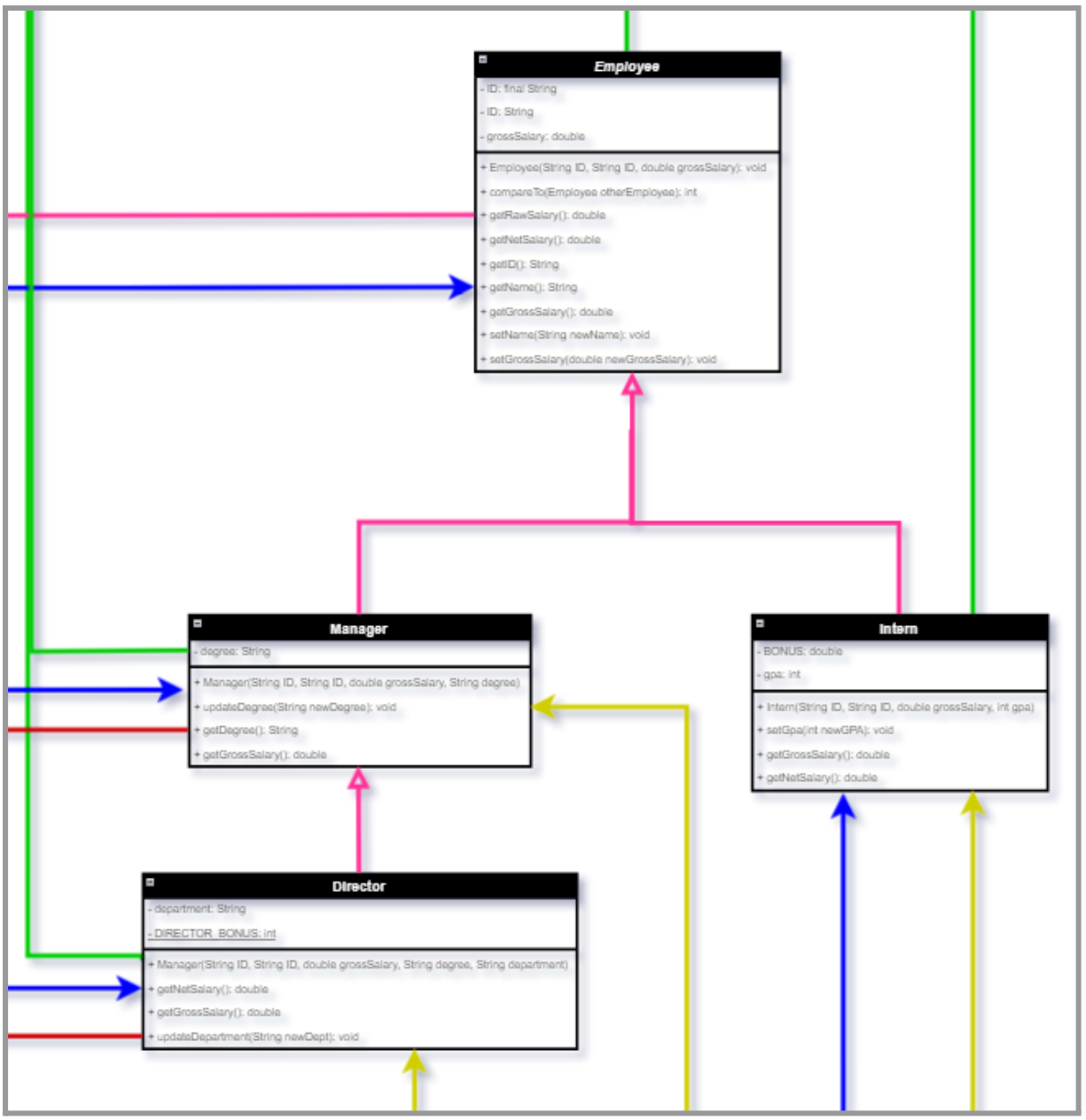
- Blue arrows: Points at the classes that are being used in Company.java
- Red arrows: Connects every class that uses Utility.java
- Green arrows: Connects every class that uses ExceptionThrower.java
- Yellow arrows: Points at every class used in EmployeeFactory.java
- Pink arrows and diamond: Displays the inheritance and composition




Company
- employees: ArrayList<Employee>
+ Company()
+ createEmployee(String employeeID, String employeeName, double grossSalary): String
+ createEmployee(String employeeID, String employeeName, double grossSalary, String degree): String
+ createEmployee(String employeeID, String employeeName, double grossSalary, String degree, String department): String
+ createEmployee(String employeeID, String employeeName, double grossSalary, int gpa): String
+ removeEmployee(String id): String
+ printEmployee(String id): String
+ printAllEmployees(): String
+ printSortedEmployees(): String
+ getNetSalary(String id): double
+ updateEmployeeName(String id, String newName): String
+ updateInternGPA(String id, int newGpa): String
+ updateManagerDegree(String id, String newDegree): String
+ updateDirectorDept(String id, String newDept): String
+ updateGrossSalary(String id, double newGrossSalary): String
+ getTotalNetSalary(): double
+ mapEachDegree(): HashMap<String, Integer>
+ formatDegrees(HashMap<String, Integer> degrees): String
+ promoteToManager(String id, String degree): String
+ promoteToDirector(String id, String degree, String department): String
+ promoteToIntern(String id, int gpa): String
- findEmployeeIndex(String id): int


EmployeeFactory
- EmployeeFactory()
+ createEmployee(String employeeID, String employeName, double grossSalary): Employee
+ createManager(String employeeID, String employeName, double grossSalary, String degree): Manager
+ createDirector(String employeeID, String employeName, double grossSalary, String degree, String department): Di
+ createIntern(String employeeID, String employeName, double grossSalary, int gpa): Intern
Utils
+ DEPARTMENTS: HashSet<String>
+ DEGREES: HashSet<String, Double>
+ EOL: String
+ truncateDouble(double value, int decimalPlaces): double
+ isStringEmpty(String string): boolean
+ registeredEmployee(String id): String
+ updatedEmployee(String id): String
+ promotedEmployee(String id, String newType): String


NegativeValueException
+ NegativeValueException()
+ NegativeValueException(String message)
StringCannotBeEmptyException
+ StringCannotBeEmptyException()
+ StringCannotBeEmptyException(String message)
ExceptionThrower
- ExceptionThrower()
+ checkEmptyId(String id): void
+ checkEmptyName(String name): void
+ checkEmptyGrossSalary(double salary): void
+ checkValidDegree(String degree): void
+ checkIfEmployeeFound(String id, int index): void
+ checkIfEmployeeRegistered(String id, int index): void
+ checkIfEmployeeRegistered(String id, int index): void
+ checkIfNoEmployees(int size): void
+ checkValidDepartment(String degree): void
+ checkValidGPA(int newGPA): void



2. Class relationships and Dependencies

- Explain where you are using Composition / Aggregation and justify your choice.

In Company.java where we declare the ArrayList of type Employee as a private attribute. This is a composition since a removal of the composite (Company) means that the component (ArrayList<Employee> employees) can't continue to exist.

Considering our objective was to work with a data structure that stores an indefinite number of our complex type Employees, we knew that an ArrayList of this particular type is a good

approach. The second question we asked ourselves was in which class we would declare ArrayList<Employee> employees in:

Declare the list in Employe.java?
We solely use this class for Employees. For instance, Manager.java and Intern.java are related to Employe.java since they are inheriting its behavior. However, this superclass only has one purpose: Define the properties and behavior of an employee. With that said it's inappropriate for it to also have the responsibility of keeping track of every single type of employee, it isn't something every employee should be able to do.

After we concluded that Employe.java shouldn't store the list, we decided to create a new class called Company.java. The connection that we made when creating this composition is the following:

- Company → Employees with different roles and skills are working on a company → Employees of different types

Another option is to define the employee-list in Utils.java, but since the guidelines for this class are to only have static methods and not create objects of it, we would have to make the list static to access it from other classes. The downside of this solution is that we can't work with several instances of a "group" or list of employees, which I consider as a disadvantageous approach due to the following reason:

- If we are developing a program that compares the success of all clothing companies in Gothenburg, we want to avoid declaring the list in a class that we can't instantiate objects from because we want to observe numerous companies (lists of employees)

From another aspect, according to the specifications we are only working with one company. Therefore I conclude this approach to be correct (when only examining the fact that we get the requested output). But If we define the list in Utils.java, it suddenly has two responsibilities:

- Taking care of general operations that are used by numerous classes in our program
- Keeping track of a group (list) of employees

My conclusion is because of these reasons that we should put the list in Company.java.

- Explain where you are using Polymorphism and justify your choice.

In Company.java we do "this.employees.add(Employee employeeToAdd) inside of 4 overloaded methods called "createEmploye()" that adds an object whose type varies from all classes in the inheritance-tree (Employee, Manager, Director, Intern). In other words, we are appending new elements to our ArrayList with the reference type Employee with objects of its subclasses.

3. Advantages of your Design

- Explain **at least 3** advantages that your design choices described in Section 2 strengthen your project.

- ArrayList of type Employee along with Polymorphism allows us to only work with one list that stores all different types of Employees. At this point, when the customer only wants 3 more specified Employees it doesn't make that much of a difference to create a separate list for every Employee. But from another perspective, if the specifications would force us to take care of for instance 40 new employees we would have to follow a long sequence of steps in our code when wanting to add or remove a specific Employee. To be more specific, we would first have to declare the new list of employees, and then implement a new method for adding or removing elements in the list. Doing this 40 times demands a lot of extra work and time that could have been saved by using one single list of the superclass along with polymorphism.

- In addition, these additional operations (when adding a new employee) require lines of code, and the more code we have in a class, the harder it becomes for us to keep track of its content and hence developing the class.

- The composition where Company.java is the composite and Employee the component improves the design of our code, because if we imagine the appearance of the code without applying composition, the structure of the code comes further away from real life abstractions (which makes the readability worse). I'm referring to the fact that we need to create a separate list for every new attribute in any subclass belonging to the inheritance tree of Employee.java to replace the composition's impact. In Company.java we would therefore have:

- ArrayList<String> ids, ArrayList<String> names, ArrayList<Double> grossSalaries, ArrayList<int> gpas, ArrayList<String> degrees, ArrayList<String> departments

  Where each ArrayList has the length of the number of employees we are working with, and employees.get(i).getGrossSalary() corresponds to grossSalaries.get(i). In the long run when the customer wants a change, it would be exhausting for us to modify our code with an unique list representing every single employee's value for a specific attribute.

Note: If you are going for a grade 5, the 3 advantages must be **in addition to** those from using Factory Method.

Benefits using Factory Methods:

- Encapsulate the creation of objects, meaning that we isolate the creation and hide the logic of creating the objects. This serves the same purpose as private attributes combined with public getters and setters: We don't want to be able to access these attributes everywhere in our program, since they only are used in a particular class. In Director.java we made the string 'department' private because no other Employee has a department, and should therefore not be able to access it.
- Factory Design Pattern - YouTube - *1:25*

- Removes the boundaries of having to know all objects and their types before running the program. If we don't know what objects our program will work with in advance and want to define what type of objects to create during runtime, we can pass a

variable as a parameter in a Factory Method to distinguish between which object to return. We achieve this by changing the value of this variable in parallel with the user's choices through the Control Flow of the program.

- The Factory Method Pattern Explained and Implemented in Java | Creational Design Patterns | Geekific - YouTube - *7:12*

- Facilitates the procedure of adding a new class that is in the inheritance tree, as an extension of the factory method occurs when adding a new type of object to return (which is related to polymorphism). With that said, this design pattern is open for extension because we don't have to change the code at multiple sections in the factory class. In EmployeeFactory.java, all we would have to do to cover the creation of a new type of Employee is to add a new method that takes its attributes in the parameters and return an object of that new type. This implies that this method favors the Open-Closed principle, as no modification anywhere else in the class is required regarding the development of the method.

- Why is the factory method design pattern more useful than having classes and calling them individually? - Software Engineering Stack Exchange - *Answered by 'pdr' 6th June 2013, about 30 upvotes, "Extensibility"*

4. Factory Method

- Explain (i) What is the Factory Class?, (ii) What is your factory method?, (iii) What is the Product?

Factory class: EmployeeFactory.java - This class is like a factory: It contains the functionality necessary to "generate" objects of Employee and its subclasses.

Product: The different types of objects that the Factory Method creates. For example, two products in our Factory Method are Manager and Intern.
http://www.cs.sjsu.edu/~pearce/modules/patterns/creational/factory.htm "Solution"

Factory Method:
The number of parameters for the types of Employees differs, which prevented us from creating a comprehensive method that can be used to return all different types of products. Therefore we defined four Factory Methods inside the Factory class, one for each type of Employee with unique parameters inside EmployeeFactory.java: (createEmploye(), createManager(), createDirector(), createIntern()

- Explain the benefits that applying Factory Method has brought to your project's design.

It separates the creation of objects from the Company.java where we previously created them favoring the Single Responsibility Principle as the new class EmployeeFactory.java only has one responsibility: Create objects. The alternative is to let Company.java have two responsibilities: Operations on employees of different types and creating them. I believe the benefits of distributing the responsibilities of classes such that each only has one are:

- More structured and readable code. I’m referring to the fact that when we read the name of a class organized in this way, we will immediately have a good idea of what it should contain.

- Classes with multiple responsibilities would result in more operations and information gathered in those classes that leads to a negative impact on the readability. In turn, it becomes more time consuming to identify a logical programming error within the class. and therefore it also becomes harder to develop the class.