Design Patterns Term Project

Problem statement and solution proposal

Özge Gürel
202151056009
December 15, 2021

Outline

The Problem

Problem Statement

Factory Pattern

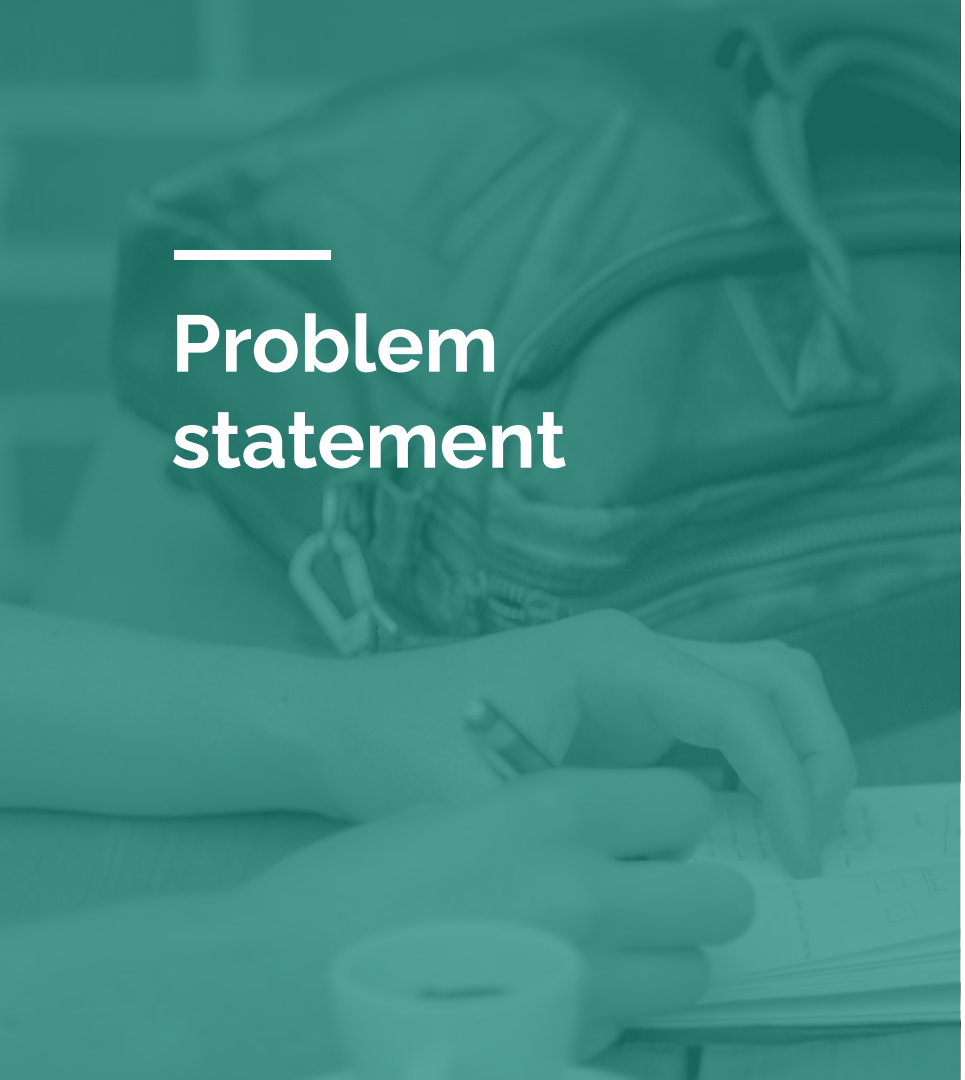
Strategy Pattern

UML Diagram

The Problem

Worker salary calculations are a detail that should be done regularly in every company and company where there are employees and employers.

Many items are taken into account when calculating a worker's salary. Both the large number of calculation items and the complexity cause the margin of error. Making mistakes in even a single digit in mathematical operations can have huge consequences. Here, technology helps to eliminate the margin of error in worker salary calculations with new generation solutions.



Problem statement

When you hire a new staff member to your business, you must open a personal file.

You save your employees with some information on your system. Some of this information are: personnel information, department, salary etc.

Increases and deductions are applied to personal file with the salary policies created by the company.

Salary can be calculated based on a fixed or percentage increase or decrease. It should be easily produced with different strategies, added to the personnel file and be calculated quickly.

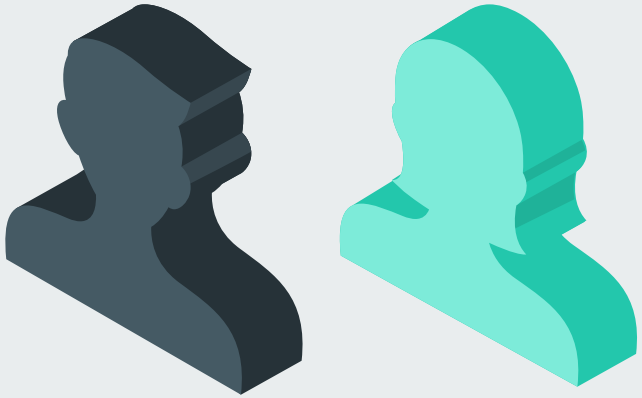
What engineers do today

Try to solve the problem.

A system is created in which information about your employees is kept and calculations can be made easily. Thanks to the practical and fast employer salary calculation program, the personnel management process is optimized.

Factory Pattern

01



A Factory Pattern says that just define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.

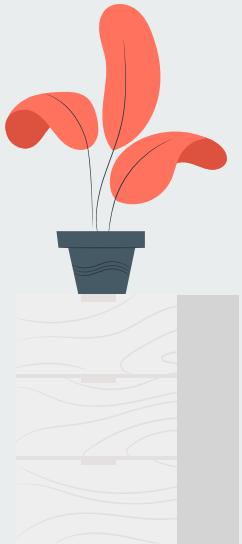
In other words, subclasses are responsible to create the instance of the class.

Factory Method pattern comes under “Creational Design Patterns”.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Factory Pattern

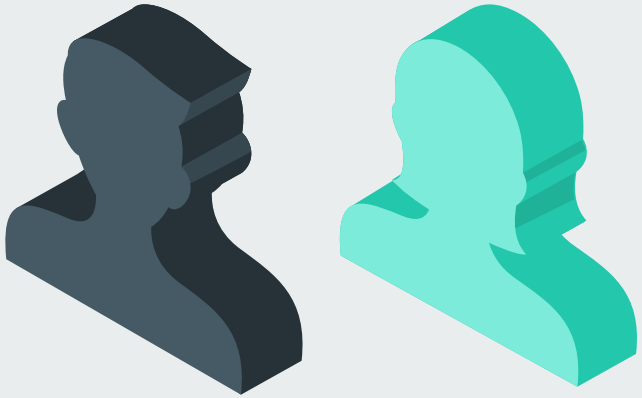
Advantages



- Factory Method Pattern allows the sub-classes to choose the type of objects to create.

- It promotes the loose-coupling by eliminating the need to bind application- specific classes into the code.
  (That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.)

Factory Pattern
Implementation



The Factory Pattern model allows new employees to be easily created in my project, taking into account the department type and promotion strategy set for the employee.

Employee

```java
public class Employee {
    public String name;
    public BigDecimal salary;
    public PromoteStrategy strategy;

    public Employee(String name, BigDecimal salary, PromoteStrategy strategy){
        this.name = name;
        this.salary = salary;
        this.strategy = strategy;
    }

    public void applyPromotion() {
        strategy.apply(this);
    }

    public void updatePromotion(PromoteStrategy strategy){
        this.strategy = strategy;
    }

    public String toString(){
        return "name:" + this.name + " salary:"+ this.salary ;
    }

}
```

Employee of Department

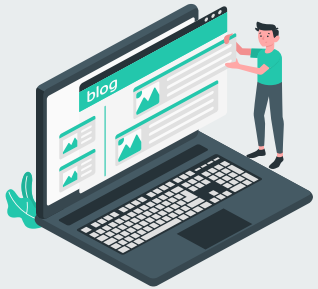
```java
public class FinancialManager extends Employee{

    public FinancialManager(String name, BigDecimal salary, PromoteStrategy strategy){
        super(name, salary, strategy);
    }

    public FinancialManager(){
        super("", new BigDecimal(0), null);
    }

}
```

```java
public class HumanPartner extends Employee{

    public HumanPartner(String name, BigDecimal salary, PromoteStrategy strategy){
        super(name, salary, strategy);
    }

    public HumanPartner(){
        super("", new BigDecimal(0), null);
    }

}
```

```java
public class Technologist extends Employee{

    public Technologist(String name, BigDecimal salary, PromoteStrategy strategy){
        super(name, salary, strategy);
    }

    public Technologist(){
        super("", new BigDecimal(0), null);
    }

}
```

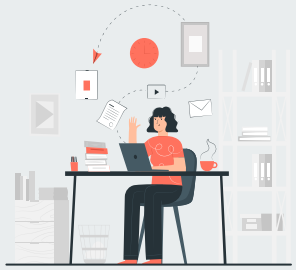
```java
public class GrowthManager extends Employee{

    public GrowthManager(String name, BigDecimal salary, PromoteStrategy strategy){
        super(name, salary, strategy);
    }

    public GrowthManager(){
        super("", new BigDecimal(0), null);
    }

}
```

EmployeeFactory

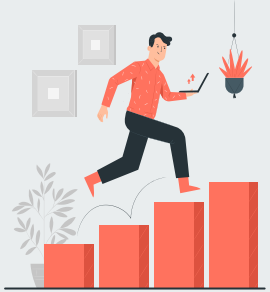


Technology

Growth



Financial



Human Partner

```
public class EmployeeFactory {

    public static Employee create(
            String name,
            BigDecimal salary,
            EmployeeDepartment department,
            PromoteStrategy strategy) {

        Employee employee = null;
        switch (department) {
            case TECHNOLOGY:
                employee = new Technologist();
                break;
            case HUMANPARTNER:
                employee = new HumanPartner();
                break;
            case FINANCE:
                employee = new FinancialManager();
                break;
            case GROWTH:
                employee = new GrowthManager();
                break;
            default:
                throw new IllegalArgumentException();
        }
        employee.strategy = strategy;
        employee.name = name;
        employee.salary = salary;
        return employee;
    }
}
```

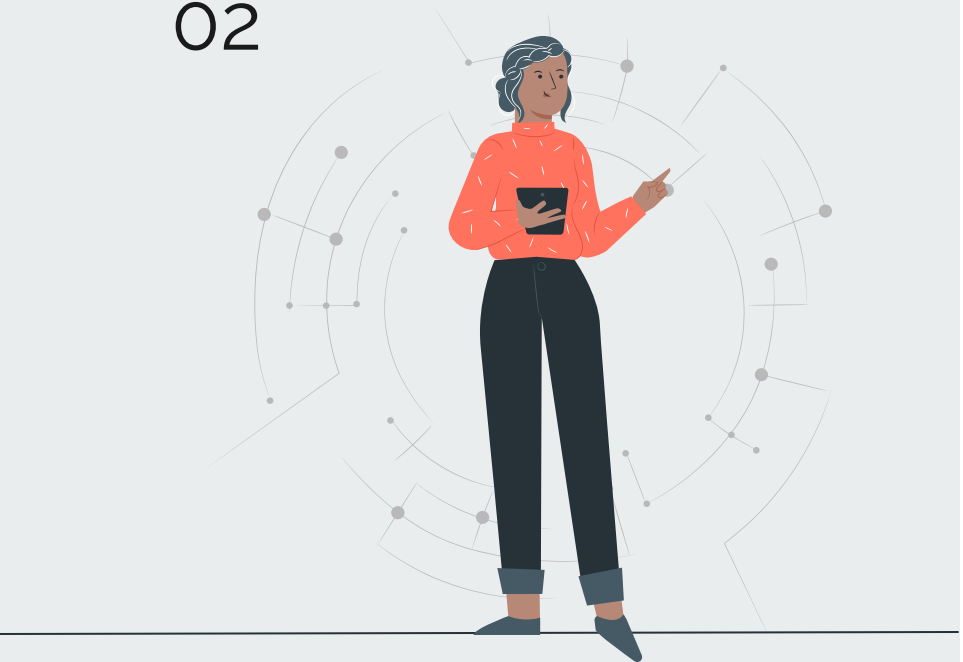
Hashmap

```
HashMap employees = new HashMap<String,Employee>();

Employee ozge = EmployeeFactory.create(
    "Özge Gürel",
    new BigDecimal(100),
    EmployeeDepartment.TECHNOLOGY,
    new FixPromoteStrategy(new BigDecimal(100))
);
System.out.println("Employee is created as "+ ozge.toString());

employees.put(ozge.name, ozge);
System.out.println("Employee is added the list " + employees);
```

```
Employee is created as name:Özge Gürel salary:100

Employee is added the list {Özge Gürel=name:Özge Gürel
salary:100}
```

Strategy Pattern

02



Strategy pattern (known as policy) is a behavioral software design pattern that enables a behavior (an algorithm) to be selected at runtime.

A Strategy defines a set of algorithms that can be used interchangeably.

Strategy Pattern

Intent

The behavior of a class to be independent from the clients that use it.

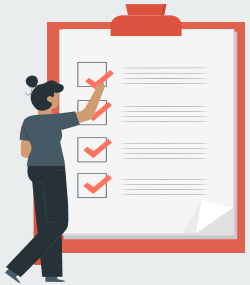
By encapsulating each algorithm and make them interchangeable.

Put the abstraction in an interface, push implementation details down to derived classes.

The behavior can change in subclasses without side effects.

Strategy Pattern

Use the SP when



Many related classes differ only in their behavior.

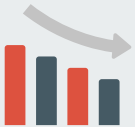
To avoid complex code.

We have multiple algorithms for a specific task and we want the client decides what strategy to be used at runtime.

Strategy Pattern
Implementation

The Strategy Pattern model enables to apply the chosen promotional behavior to the worker. And it helps worker information, easily adapt to change. Also, the promotional strategy switches can be implemented quickly.

Promote Strategy



```java
public interface PromoteStrategy {
    Employee apply(Employee employee);
}
```

Promote Strategy

```java
public class FixPromoteStrategy implements PromoteStrategy {
    BigDecimal fixAmount = new BigDecimal(0);

    public FixPromoteStrategy(BigDecimal fixAmount){
        this.fixAmount = fixAmount;
    }

    @Override
    public Employee apply(Employee employee) {
        employee.salary = employee.salary.add(fixAmount);
        return employee;
    }
}
```

```java
public class FixPenaltyStrategy implements PromoteStrategy {
    BigDecimal fixAmount = new BigDecimal(0);

    public FixPenaltyStrategy(BigDecimal fixAmount){
        this.fixAmount = fixAmount;
    }

    @Override
    public Employee apply(Employee employee) {
        employee.salary = employee.salary.subtract(fixAmount);
        return employee;
    }
}
```

```java
public class PercentagePenaltyStrategy implements PromoteStrategy{
    BigDecimal percentAmount = new BigDecimal(0);

    public PercentagePenaltyStrategy(BigDecimal percentAmount){
        this.percentAmount = percentAmount;

    }

    @Override
    public Employee apply(Employee employee) {
        BigDecimal hundred = new BigDecimal(100);
        BigDecimal multipleNumber = hundred.subtract(percentAmount).divide(hundred);
        employee.salary = employee.salary.multiply(multipleNumber);
        return employee;
    }
}
```

```java
public class PercentagePromoteStrategy implements PromoteStrategy{
    BigDecimal percentAmount = new BigDecimal(0);

    public PercentagePromoteStrategy(BigDecimal percentAmount){
        this.percentAmount = percentAmount;

    }

    @Override
    public Employee apply(Employee employee) {
        BigDecimal hundred = new BigDecimal(100);
        BigDecimal multipleNumber = percentAmount.add(hundred).divide(hundred);
        employee.salary = employee.salary.multiply(multipleNumber);
        return employee;
    }
}
```

Promote Strategy

Özge

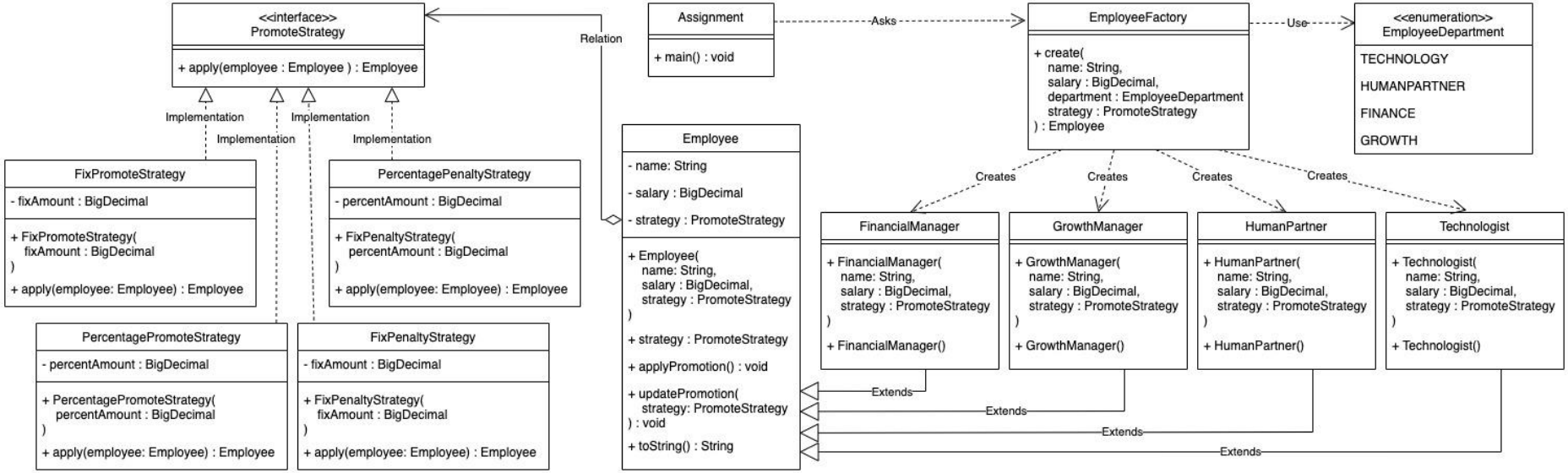
Technologist

```
Employee ozge = EmployeeFactory.create(
    "Özge Gürel",
    new BigDecimal(100),
    EmployeeDepartment.TECHNOLOGY,
    new FixPromoteStrategy(new BigDecimal(100))
);
System.out.println("Employee is created as "+ ozge.toString());

ozge.applyPromotion();
System.out.println("Employee has promoted as "+ ozge.toString());

ozge.applyPromotion();
System.out.println("Employee has promoted as "+ ozge.toString());

ozge.updatePromotion(new PercentagePromoteStrategy(new BigDecimal(20)));
System.out.println("Employee has changed promote strategy: "+ ozge.toString());

ozge.applyPromotion();
System.out.println("Employee has promoted as "+ ozge.toString());

ozge.updatePromotion(new FixPenaltyStrategy(new BigDecimal(100)));
System.out.println("Employee has changed promote strategy: "+ ozge.toString());

ozge.applyPromotion();
System.out.println("Salary reduction was made for this employee: "+ ozge.toString());
```

```
Employee is created as name:Özge Gürel salary:100
Employee has promoted as name:Özge Gürel salary:200
Employee has promoted as name:Özge Gürel salary:300
Employee has changed promote strategy: name:Özge Gürel salary:300
Employee has promoted as name:Özge Gürel salary:360.0
Employee has changed promote strategy: name:Özge Gürel
salary:360.0
Salary reduction was made for this employee: name:Özge Gürel
salary:260.0
```

UML Diagram

Employer Salary Calculator UML Diagram



Questions?

Thank you.