Lab VIII.

Objetivos

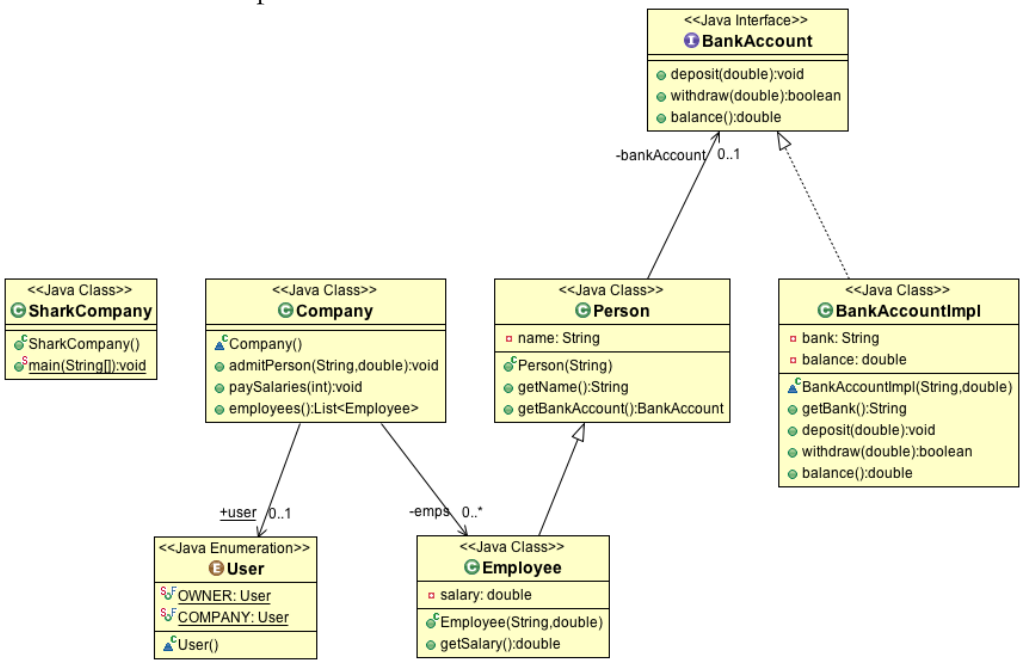
Os objetivos deste trabalho são:

- Utilizar padrões estruturais (i.e., Adapter, Facade, Proxy, Flyweight, etc.) para resolver casos práticos.
- Aplicar boas práticas de programação por padrões

Nota: Para além do código no codeUA, inclua também um ficheiro PDF ou PNG com o diagrama de classes da solução final (pode usar o UMLet, por exemplo, ou um plugin Eclipse/Netbeans).

VIII.1 Gestão de acesso a conta bancária

Considere o programa seguinte que pretende gerir os pagamentos dos salários de funcionários de uma empresa.




```
interface BankAccount {
     void deposit(double amount);
     boolean withdraw(double amount);
     double balance();
}
class BankAccountImpl implements BankAccount {
     private String bank;
     private double balance;
     BankAccountImpl(String bank, double initialDeposit) {
           this.bank = bank;
           balance = initialDeposit;
     }
     public String getBank() {
           return bank;
     }
     @Override public void deposit(double amount) {
           balance += amount;
```

```
    }
    @Override public boolean withdraw(double amount) {
          if (amount > balance )
              return false;
          balance -= amount;
          return true;
    }
    @Override public double balance() {
          return balance;
    }
}

class Person {
    private String name;
    private BankAccount bankAccount;

    public Person(String n) {
          name = n;
          bankAccount = new BankAccountImpl("PeDeMeia", 0);
    }
    public String getName() {
          return name;
    }
    public BankAccount getBankAccount() {
          return bankAccount;
    }
}

class Employee extends Person {
    private double salary;

    public Employee(String n, double s) {
          super(n);
          salary = s;
    }
    public double getSalary() {
          return salary;
    }
}

enum User { OWNER, COMPANY  }

class Company {
    public static User user;
    private List<Employee> emps = new ArrayList<>();

    public void admitPerson(String name, double salary) {
          Employee e = new Employee(name, salary);
          emps.add(e);
    }
    public void paySalaries(int month) {
          for (Employee e : emps) {
              BankAccount ba = e.getBankAccount();
              ba.deposit(e.getSalary());
          }
    }
    public List<Employee> employees() {
          return Collections.unmodifiableList(emps);
    }
}
```

```
public class SharkCompany {

    public static void main(String[] args) {
          Company shark = new Company();
          Company.user = User.COMPANY;
          shark.admitPerson("Maria Silva", 1000);
          shark.admitPerson("Manuel Pereira", 900);
          shark.admitPerson("Aurora Machado", 1200);
          shark.admitPerson("Augusto Lima", 1100);
          List<Employee> sharkEmps = shark.employees();
          for (Employee e : sharkEmps)
              // "talking to strangers", but this is not a normal case
              System.out.println(e.getBankAccount().balance());
          shark.paySalaries(1);
          for (Employee e : sharkEmps) {
              e.getBankAccount().withdraw(500);
              System.out.println(e.getBankAccount().balance());
    }
    }
}
```

Na implementação atual é possível que a empresa aceda aos dados privados da conta bancária de cada pessoa.

a) Construa uma solução que permita ao funcionário impedir a empresa de ter acesso aos métodos *withdraw* e *balance* da sua conta bancária, mantendo ao mesmo tempo a possibilidade de ele próprio aceder a tudo. Utilize a variável *Company.user* para simular o perfil do utilizador. Nesta solução não pode alterar as classes existentes (apenas modificar ligeiramente a classe *Person*).

b) Considerando que as classes *Person* e *Employee* fazem parte de domínios distintos crie uma nova versão do programa (*SharkCompany2*) onde *Employee* não herda de *Person* e o acesso à conta bancária fica limitado à classe *Employee*. Assim deixa de ser possível as funções cliente (*main* por exemplo) usarem expressões como *e.getBankAccount().balance()* (princípio *"Don't talk to stranger"*). Note que esta solução não resolve *per si* o problema identificado em a) uma vez que, se nada for feito, a classe *Employee* (classe da empresa) pode aceder a todos os métodos de *BankAccount*. Exemplo possível para a função *main*:

```
public class SharkCompany {

    public static void main(String[] args) {
          Person[] persons = { new Person("Maria Silva"),
                      new Person("Manuel Pereira"),
                      new Person("Aurora Machado"),
                      new Person("Augusto Lima")   };
          Company shark = new Company();
          Company.user = User.COMPANY;
          shark.admitEmployee(persons[0], 1000);
          shark.admitEmployee(persons[1], 900);
          shark.admitEmployee(persons[2], 1200);
          shark.admitEmployee(persons[3], 1100);
          List<Employee> sharkEmps = shark.employees();
          for (Employee e : sharkEmps)
                System.out.println(e.getSalary());
          shark.paySalaries(1);
    }
}
```

VIII.2 SharkCompany with a Facade

Com base na implementação anterior pretende-se agora criar uma Facade (pode usar a classe *Company* e o método *admitEmployee* para evitar criar uma nova classe) que garanta que quando um novo funcionário é admitido, para além do registo na empresa, são igualmente invocados os seguintes serviços:

1. Registo na segurança social (e.g. *SocialSecurity.regist(person)*)
2. Registo na seguradora (e.g. *Insurance.regist(person)*)
3. Criação de um cartão de funcionário
4. Autorização para use de parque automóvel caso o salário seja superior à média (e.g. *Parking.allow(person)*)

Construa entidades e métodos adequados a este problema. Note que o enfâse é na construção da *facade* e menos na construção de métodos nas novas classes que vai necessitar.