UAST
Unidade Acadêmica de Serra Talhada - PE
UFRPE
Desde 2006
BSi
Bacharelado em Sistemas de Informação

MPOO

Site: https://sigs.ufrpe.br/sigaa/ava/index.jsf

Disciplina: Modelagem e Programação Orientada a Objetos (MPOO)

Profº: Richarlyson D'Emery

LISTA DE EXERCÍCIOS VII

Leia atentamente as instruções gerais:

- No Eclipse crie um novo **projeto chamado br.edu.mpoo.listaVII.SeuNomeSobrenome**, o qual deverá ter **pastas de pacotes** para cada questão: questao1, questao2, e assim sucessivamente, contendo todas as respostas da lista.
- A lista envolve questões práticas e conceituais.
- Quando a questão envolver uma discussão teórica utilize um arquivo .txt (Menu File -> Submenu New -> Opção File), por exemplo, questao1.txt

Fique atento!

Prezado aluno, esta é a lista de exercícios é relativa ao assunto de: "Herança" e "Polimorfismo de Objetos".

Você Sabia?

Uso de **downcast** em POO: Uma solução ao comportamento de Herança.

```
// SuperClasse.java
public class SuperClasse {
    private int atr_SuperClasse;

        public int getAtr_SuperClasse() { return atr_SuperClasse; }

        public void setAtr_SuperClasse(int atr_SuperClasse) {
                this.atr_SuperClasse = atr_SuperClasse;
        }
}
// SubClasse.java
public class SubClasse extends SuperClasse{
        private int atr_SubClasse;

        public int getAtr_SubClasse() { return atr_SubClasse; }

        public void setAtr_SubClasse(int atr_SubClasse) {
                this.atr_SubClasse = atr_SubClasse;
        }
}
// App.java
public class App {
        public static void main(String[] args) {
                SubClasse subClasse = new SubClasse();
                System.out.println(subClasse.getAtr_SuperClasse());
                System.out.println(subClasse.getAtr_SubClasse());
/*
* Questionamento: e se subClasse fosse do tipo SuperClasse? como acessar o atr_SubClasse?
* Solução: Usar downcast.
*/
                SuperClasse subClasse2 = new SubClasse();
                System.out.println(subClasse2.getAtr_SuperClasse());
                System.out.println(((SubClasse)subClasse2).getAtr_SubClasse());//solução
        }
}
```

Saiba Mais!

Se, em tempo de execução, a referência de um objeto de subclasse tiver sido atribuída a uma variável de uma das suas superclasses diretas ou indiretas, é aceitável fazer **downcast** da referência armazenada nessa variável de superclasse de volta a uma referência do tipo da subclasse. Antes de realizar essa coerção, utilize o operador **instanceof** para assegurar que o objeto é de fato um objeto de um tipo de subclasse apropriado.

```
SubClasse subClasse = new SubClasse();
SuperClasse superClasse;
superClasse = subClasse;
if(superClasse instanceof SubClasse) //PREVINA-SE: USE instanceof
       System.out.println(((SubClasse) superClasse).getAtr_SubClasse());
```

Mas atenção! 

um erro comum de programação atribuir uma variável de superclasse a uma variável de subclasse (sem uma coerção explícita) é um erro de compilação.

```
SubClasse subClasse = new SuperClasse();
```

1) Responda V se verdadeiro ou F se Falso. Justifique se falso.

1.1) ( ) O objeto de uma subclasse pode ser tratado como um objeto de sua superclasse, mas o contrário não é verdadeiro.

1.2) ( ) Quando um método de uma superclasse é inadequado para a subclasse, o programador deve sobrescrever esse método na subclasse.

1.3) ( ) Uma superclasse representa um número maior de membros que sua subclasse

1.4) ( ) O objeto de uma subclasse também é um objeto da superclasse dessa subclasse.

Fique Atento!

Utilizando `foreach` e **instanceof** podemos reformular App, evitando erros de **downcast** e utilizar a vinculação dinâmica de `toString` em App:

```
//SuperClasse.java
public class SuperClasse {
       private int atr_SuperClasse;

       public int getAtr_SuperClasse() { return atr_SuperClasse;}

       public void setAtr_SuperClasse(int atr_SuperClasse) {
              this.atr_SuperClasse = atr_SuperClasse;
       }

       @Override
       public String toString() {
              return "[atr_SuperClasse=" + atr_SuperClasse + "]";
       }
 }

//SubClasse.java
public class SubClasse extends SuperClasse{
       private int atr_SubClasse;

       public int getAtr_SubClasse() { return atr_SubClasse;}

       public void setAtr_SubClasse(int atr_SubClasse) {
              this.atr_SubClasse = atr_SubClasse;
       }

       @Override
       public String toString() {
              return "[atr_SubClasse=" + atr_SubClasse + "," + super.toString() + "]";
       }
}
 //continua...
```

```java
// App.java
import java.util.ArrayList;

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

		// Usando ArrayList:
		ArrayList<SuperClasse> superClasses = new ArrayList<SuperClasse>();
		superClasses.add(new SuperClasse());
		superClasses.add(new SubClasse());

		// Usando foreach e instanceof:
		int cont=0;
		for (SuperClasse superClasseCurrente:superClasses){
			System.out.println("Element: [" + cont + "]");
			if(superClasseCurrente instanceof SuperClasse)
				System.out.println(superClasseCurrente.getAtr_SuperClasse());
			if(superClasseCurrente instanceof SubClasse)
				System.out.println(((SubClasse)superClasseCurrente).getAtr_SubClasse());
			cont++;
		}

		// Usando vinculação dinâmica com toString():
		cont=0;
		for (SuperClasse superClasseCurrente:superClasses){
			System.out.println("Element: [" + cont + "]");
			System.out.println(superClasseCurrente.toString());
			cont++;
		}
	}
}
```

Lembre-se:

Devido ao polimorfismo de objetos, o processo de vinculação dinâmica faz com que toString () seja resolvido em tempo de execução (em vez de em tempo de compilação) de acordo com o objeto que o invoca.

Fique Mais Atento!

Mas não é necessário explicitar `toString`:

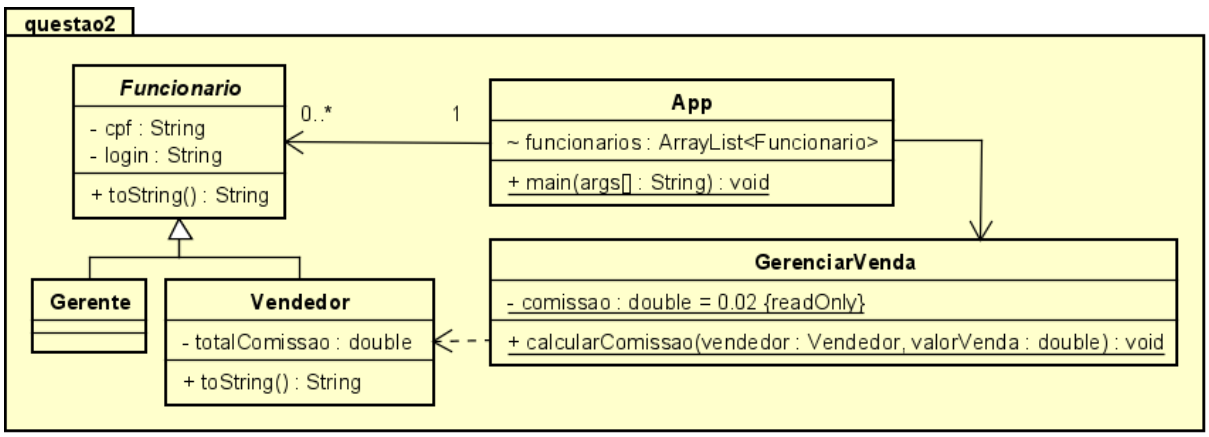
```java
	// Usando vinculação dinâmica com toString():
		cont=0;
		for (SuperClasse superClasseCurrente:superClasses){
			System.out.println("Element: [" + cont + "]");
			System.out.println(superClasseCurrente);
			cont++;
		}
	}
}
```

Lembre-se:

Mesmo que:
`superClasseCurrente.toString()`

Devido ao polimorfismo de objetos, o processo de vinculação dinâmica faz com que toString () seja resolvido em tempo de execução (em vez de em tempo de compilação) de acordo com o objeto que o invoca.

2) O diagrama de classes abaixo ilustra a vinculação dinâmica da chamada de toString para o polimorfismo entre as especializações Vendedor e Gerente e a generalização Funcionário. A codificação está demonstrada na pasta de pacote questao2 do projeto disponibilizado juntamente com esta Lista.



Na seção “Fique Atento!” foi apresentado um exemplo de polimorfismo de herança/objetos no qual realiza a vinculação dinâmica da chamada do método `toString()`. Para isso, o método foi reescrito pela sobreposição de métodos `toString` de `Object`.

2.1) Observe o erro de semântica na saída da codificação, uma vez que ao invés de “Gerente” tem-se “Funcionário”. Realize a devida sobreposição de maneira a garantir a saída correta de acordo com a função definida através da Herança.

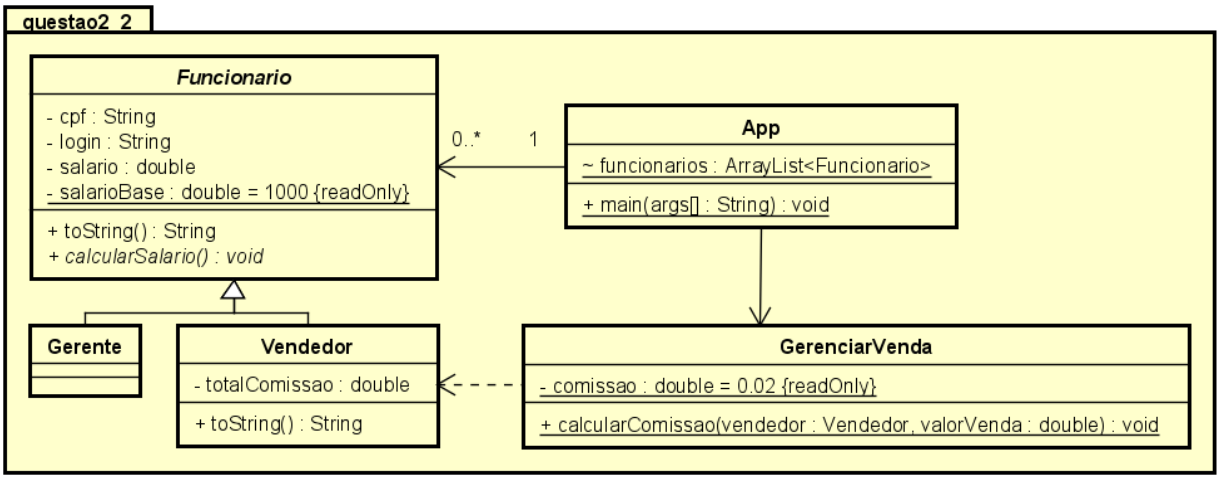
```
Funcionario [cpf=222.222.222-22, login=Ana Souza]
Vendedor [totalVendas=600.0, Funcionario [cpf=111.111.111-11, login=João Silva]]
```

2.2) Assumindo o fato de que um funcionário possui um salário e este deve ser definido a partir da sua função: um gerente recebe 5 vezes um salário base de R$1.000,00; enquanto um vendedor tem ao seu salário base a adição de uma comissão sobre as vendas realizadas. A partir da codificação disponibilizada, implemente em Java o diagrama de classes abaixo e exiba os dados (inclusive o salário) do Gerente e Vendedor (com uma venda de R$ 30.000,00) conforme ilustrado na saída:.

Saída:

```
Gerente [cpf=222.222.222-22, login=Ana Souza, salario=5000.0]
Vendedor [totalVendas=600.0, cpf=111.111.111-11, login=João Silva, salario=1600.0]
```

Diagrama de classes:

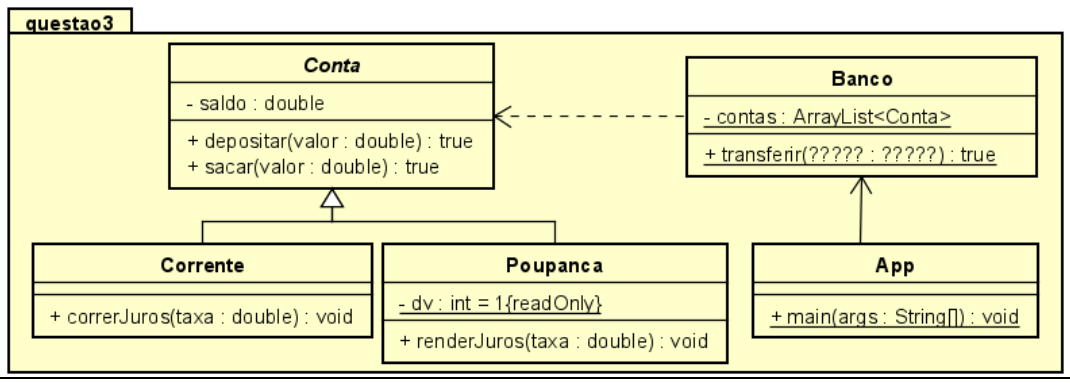


3) Comumente o brasileiro possui dois tipos de contas: uma Corrente e uma Poupança. Um banco possui diversas contas e pode realizar a transferência entre duas contas distintas (Corrente para Corrente, Corrente para Poupança, Poupança para Corrente e Poupança para Poupança). Sendo assim, quantos métodos para realizar transferências entre contas são necessários para a classe Banco? Responda codificando o problema em Java.

Mas antes, analise as seguintes regras de negócio:

- RN01 – O saldo de uma conta só pode ser movimentado pelos métodos depositar (adiciona um valor ao saldo) e sacar (retira um valor do saldo);
- RN02 – Um saque só pode ser realizado se o saldo de uma conta for igual ao maior ao valor a ser retirado;
- RN03 – Conta corrente possui o comportamento de correr juros que pode ter uma taxa a ser incidida sobre o saldo de maneira a diminuir o valor do saldo;
- RN04 – Conta poupança possui o comportamento render juros que pode ter uma taxa a ser incidida sobre o saldo de maneira a aumentar o valor do saldo;

É diagrama do sistema:



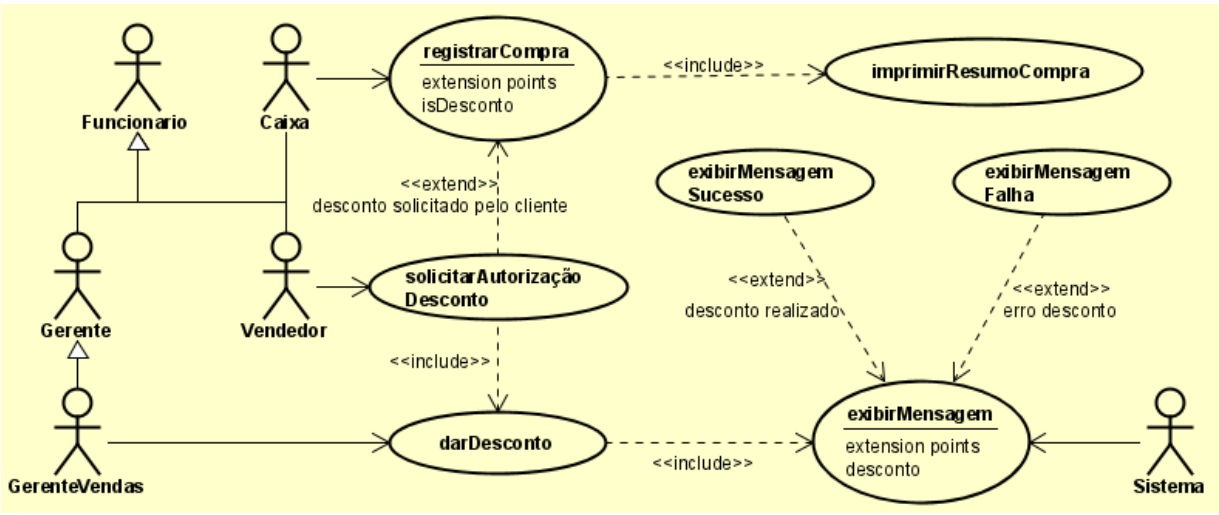
Então codifique uma aplicação em que se demonstrada:

- A criação de duas contas correntes e duas poupanças (Atribua saldos distintos a essas contas);
- Implemente a problemática de transferência entre contas (método `transferir`);
- Demonstre as transferências entre as contas: Corrente para Corrente, Corrente para Poupança, Poupança para Corrente e Poupança para Poupança;
- Demonstre o rendimento de juros e a operação de correr juros em duas contas.

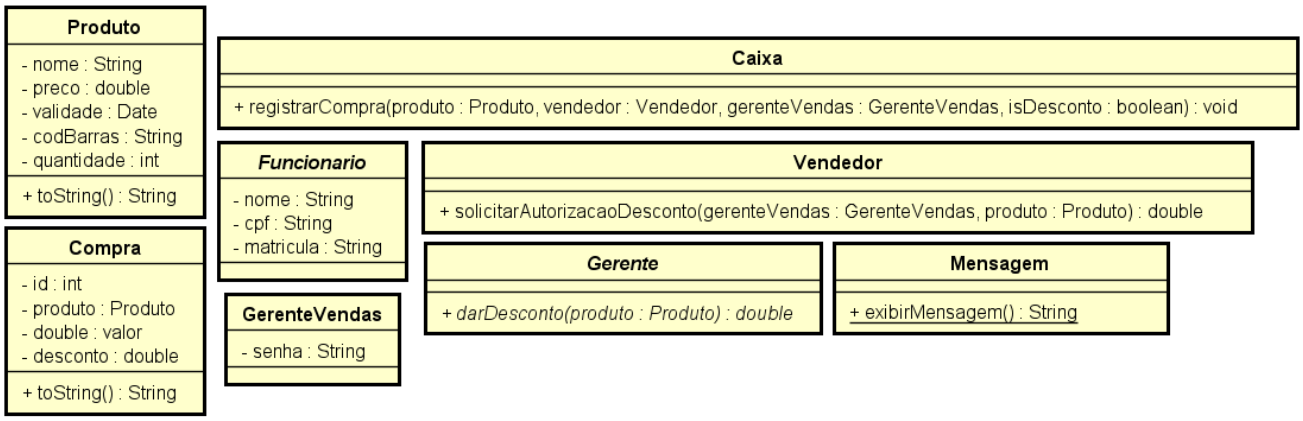
Desafio

Você, aluno de MPOO, está experienciando situações-problemas do universo de desenvolvimento de software e começará a ser desafiado a solucionar novos problemas a partir de conhecimentos de POO.

4) **(Desafio)** Uma Empresa solicitou a um de seus programadores (de codinome *mustela putórius furo* – "O Furão") que resolvesse uma demanda de funcionalidades de seu sistema criada pelo setor de venda da empresa. Cada ator detém comportamento(s) específico(s) conforme sua função ilustrada no diagrama de use case abaixo:



Esboço de diagrama do projeto sem relacionamentos:



Sabe-se que o sistema da empresa deve ser codificado em Java e também deve incluir um diagrama de classes com seus devidos relacionamentos. Antes de responder, analise as situações. Em todas essas, deve-se respeitar as seguintes regras de negócios:

- RN01 – um funcionário só pode executar os comportamentos definidos;
- RN02 – a codificação deve aproveitar comportamentos já definidos, evitando a duplicidade de programação;
- RN03 – uma compra tem identificação autoincrementável;
- RN04 – um desconto só é atribuído a uma compra se for solicitado verbalmente por um cliente no momento do registro de uma compra;
- RN05 – Um desconto ou solicitação de autorização de desconto só poderá ser executado por um funcionário válido; e
- RN06 – Em nenhuma hipótese deve-se alterar o valor de um produto.

Ilustre em uma aplicação: (i) a compra de um produto em que o cliente solicita verbalmente um desconto; e (ii) a compra de um produto em que o cliente não solicita um desconto. Em ambos os casos deve-se exibir em console o resumo da compra:

```
Console
Compra [id=1, Loção para Barba(COD01111), R$ 100.0, Desconto: R$0.0, Total Compra: R$ 100.0]
Compra [id=2, Loção para Barba(COD01111), R$ 100.0, Desconto: R$10.0, Total Compra: R$ 90.0]
```

5) O programador "O Furão" resolveu criar uma aplicação para ilustrar o acesso aos atributos de um gerente de vendas:

```
public class App {

    public static void main(String[] args) {

        GerenteVendas gerente1Venda = new GerenteVendas("Ermenegildo Silva", "111.111.111-11", "GERVEN001", "aAbBcCdD");
        Funcionario gerente2Venda = new GerenteVendas("Pregentino Santos", "222.222.222-22", "GERVEN002", "UrsoPAndA");
        System.out.println(gerente1Venda.getSenha());
        System.out.println(gerente2Venda.get);
    }
}
```

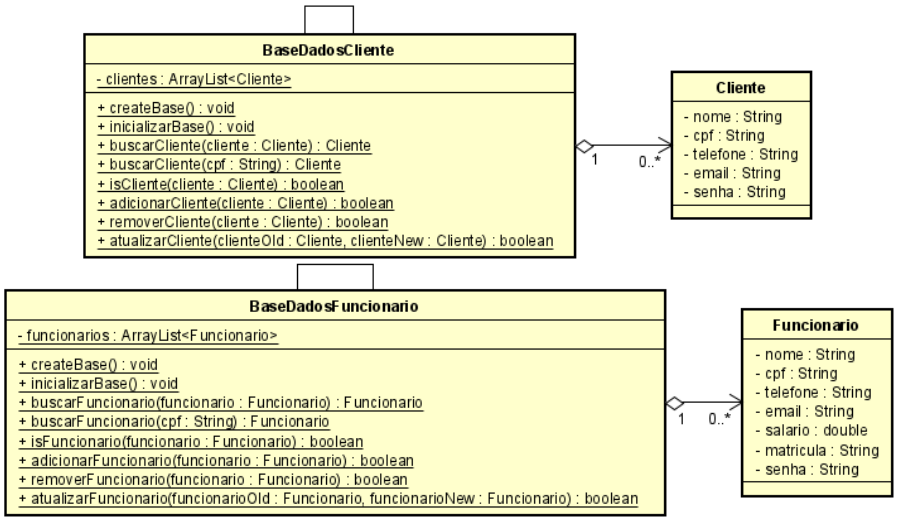
```
getCpf() : String - Funcionario
getMatricula() : String - Funcionario
getNome() : String - Funcionario
getClass() : Class<?> - Object
Press 'Ctrl+Space' to show Template Proposals
```

Entretanto, não soube explicar o que aconteceu quando tentou acessar o método `getSenha()` para exibir a senha do gerente Pregentino Santos. A partir dos conceitos de Orientação a Objetos:

5.1) Explique o que aconteceu.

5.2) Sem redefinir a definição para a instância de `gerente2Venda`, como "O Furão" poderia acessar o método `getSenha()`?

6) Na **Lista de Exercícios IV**, vimos a demanda de um contratante de um sistema de cadastro para a empresa MPOOSoftware LTDA. O Scrum Master de MPOOSoftware LTDA solicitou a um de seus programadores (de codinome *mustela putórius furo* – "O Furão") a implementação dos diagramas de classes apresentados utilizavam de duas bases: BaseDadosCliente e BaseDadosFuncionario.

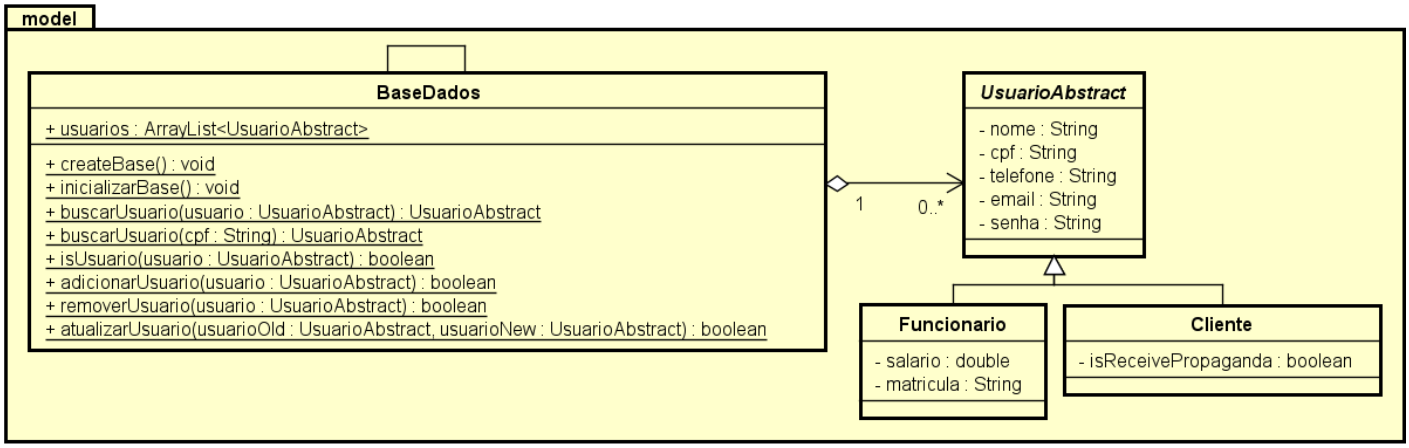


"O Furão", após treinamento em POO, aprendeu sobre polimorfismo de objetos. Após analisar os diagramas e as regras de negócios:

- RN01 – um cliente ou funcionário é identificado pelo seu cpf;
- RN02 – um cliente ou funcionário só é adicionado em uma base se tiver um cpf válido;
- RN03 – um cliente ou funcionário só poderá ser cadastrado uma única vez;
- RN04 – para entrar no sistema um cliente ou funcionário deverá informar seu login (email ou cpf) e senha;
- RN05 – a senha de um cliente ou funcionário deverá ser criada na primeira utilização do sistema;
- RN06 – a senha de um cliente ou funcionário deverá ter pelo menos 6 dígitos.

- RN07 – a codificação deve aproveitar comportamentos já definidos, evitando a duplicidade de programação; e
- **RN08 – a empresa só envia propaganda se o cliente permitir recebê-la.**

Propôs uma melhoria para o sistema (diagrama abaixo) que foi aceita por seu Scrum Master, fazendo com que passasse a ter uma única base com uso de polimorfismo:

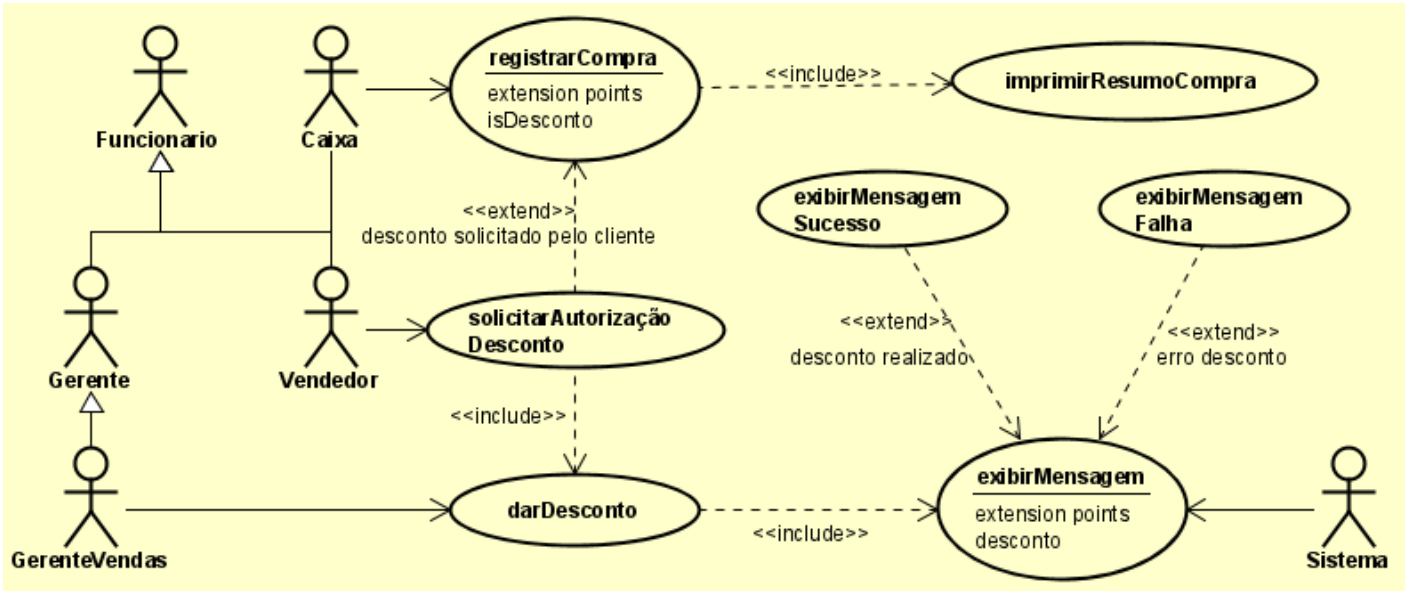


A partir da nova proposta, modifique o sistema e ilustre em uma aplicação o funcionamento dos serviços da base para dois clientes e dois funcionários (*criação de objetos e utilização da base por linha de comando no main*).

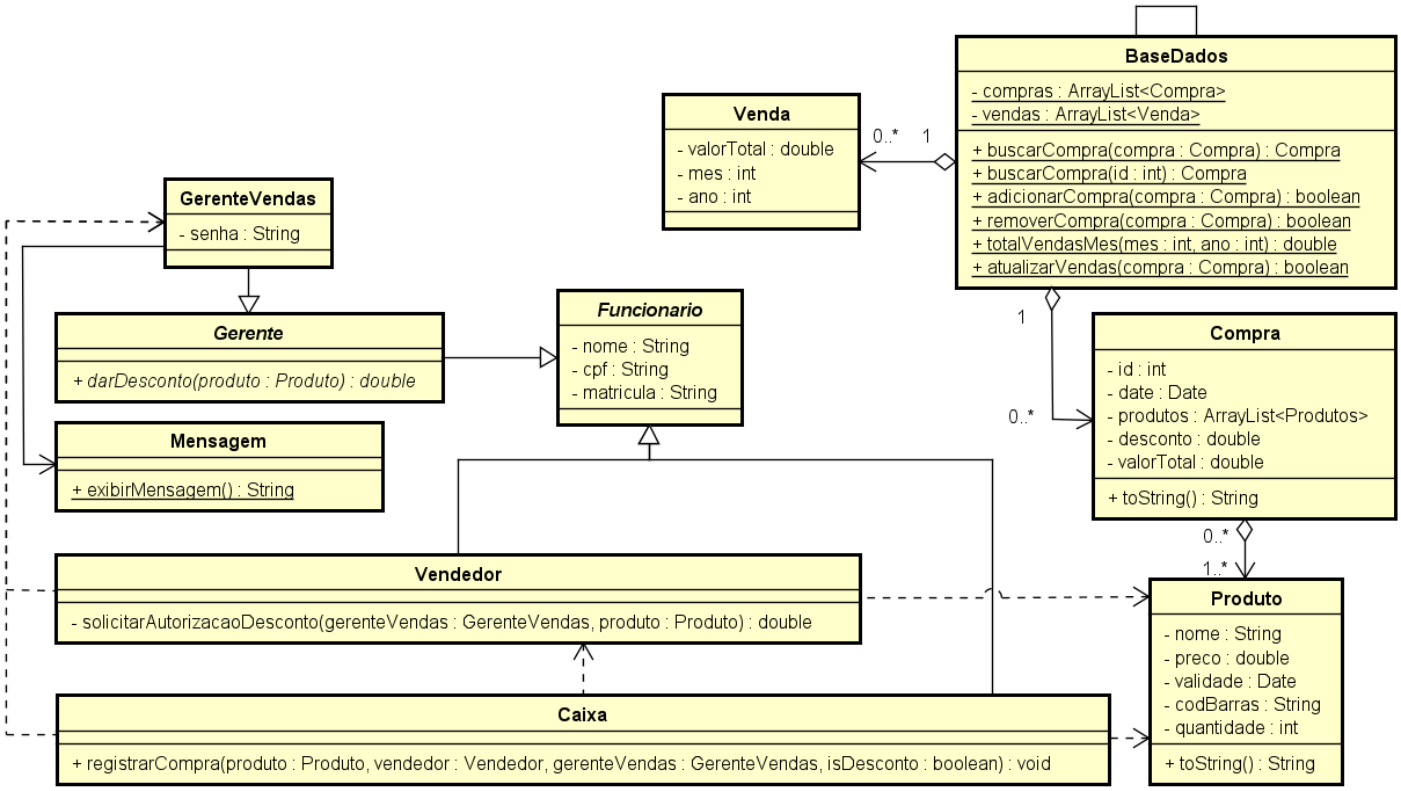
7) O contratante do sistema da questão 6) observou a necessidade de manter em sua base dados de pessoas que poderiam ser cliente da empresa, ou seja, ainda não são usuários. Sendo assim, modifique o diagrama proposto que utiliza polimorfismo de objetos, herança e classe abstrata e o codifique para atender a nova demanda.

Mão na Massa: Desafio!

8) **(Desafio)** Uma Empresa solicitou a um de seus programadores (de codinome *mustela putórius furo* – "O Furão") que resolvesse uma demanda de funcionalidades de seu sistema criada pelo setor de venda da empresa. Cada ator detém comportamento(s) específico(s) conforme sua função ilustrada no diagrama de use case abaixo:



Sabe-se que o sistema da empresa deve ser codificado em Java e também deve refletir o diagrama de classes abaixo:



Antes de responder, analise as situações contendo as seguintes regras de negócios:

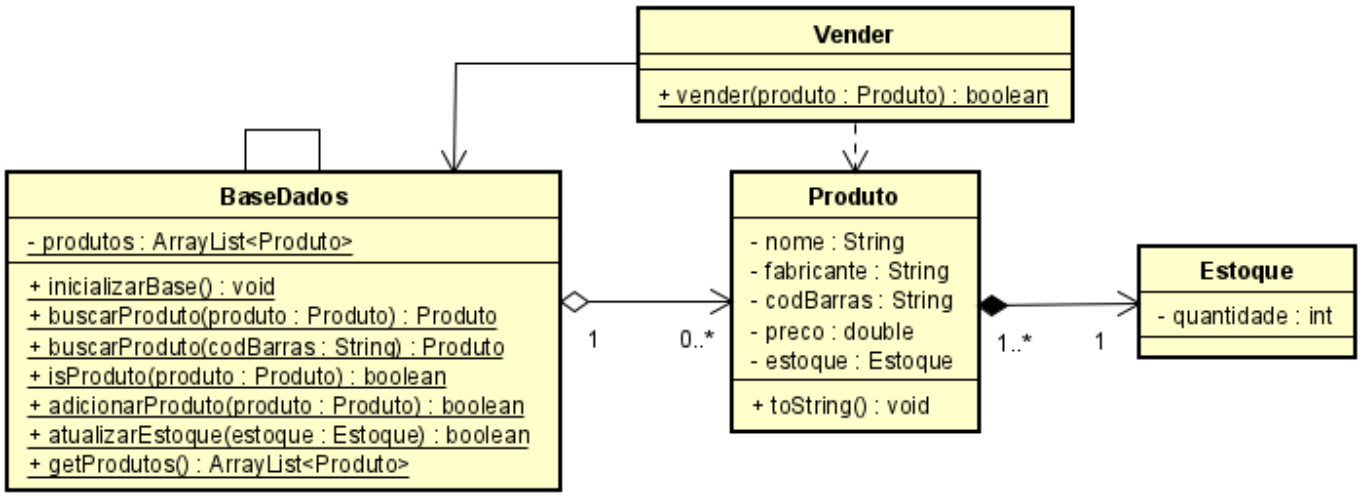
- RN01 – uma compra possui id autoincrementável;
- RN02 – a data de uma compra utiliza a data do sistema de acordo com o momento em que é realizada;
- RN03 – toda compra é registrada na base;
- RN04 – toda nova compra deve atualizar o valor de vendas de um mês;
- RN05 – toda compra cancelada deve atualizar o valor de vendas de um mês;
- RN06 – um funcionário só pode executar os comportamentos definidos;
- RN07 – a codificação deve aproveitar comportamentos já definidos, evitando a duplicidade de programação;
- RN08 – uma compra tem identificação autoincrementável;
- RN09 – um desconto só é atribuído a uma compra se for solicitado verbalmente por um cliente no momento do registro de uma compra;
- RN10 – Um desconto ou solicitação de autorização de desconto só poderá ser executado por um funcionário válido; e
- RN11 – Em nenhuma hipótese deve-se alterar o valor de um produto.

Ilustre em uma aplicação: (i) a compra de um produto em que o cliente solicita verbalmente um desconto; (ii) a compra de um produto em que o cliente não solicita um desconto; e (iii) a atualização das vendas para uma compra realizada e para outra cancelada. Em todos os casos deve-se exibir em console o resumo da compra e das vendas.

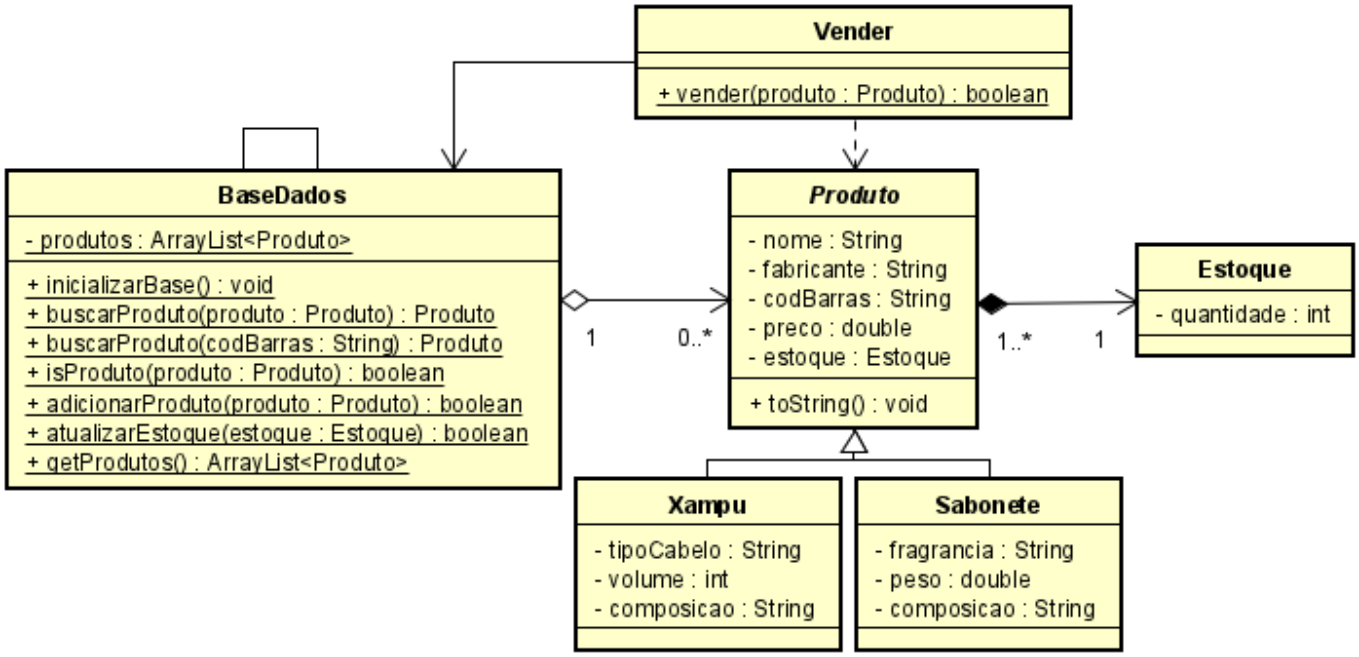
//continua...

Mão na Massa: Desafio – Integrando Conceitos!

9) **(Desafio)** Na **Lista de Exercícios V**, vimos o problema de armazenar em uma base de dados o tipo de dados Produto e sua quantidade disponível em estoque:



Entretanto, é possível especificar quais os produtos deverão ser armazenados. Exercite a integração dos conceitos de **Classe Abstrata**, **Herança** e **Composição**, a partir do diagrama abaixo:



São regras de negócios:

- RN01 – um produto é identificado pelo codBarras;
- RN02 – um produto só poderá ser cadastrado uma única vez;
- RN03 – a quantidade de produto poder ser vista pelo atributo quantidade;
- RN04 – atualizarEstoque atualiza a quantidade de produtos disponíveis; e
- RN05 – um produto quando vendido tem seu estoque atualizado.

A partir dessas características, implemente o sistema em Java. Demonstre em uma aplicação:

- A criação de diversos produtos;
- A atualização de estoque dos produtos criados; e
- A venda de alguns produtos.