2 – Não apresentam, pois, todas as subclasses são exclusivas e não tem possibilidade de se transformar em outra classe, sendo assim não há necessidade de criar várias possibilidades de subclasse.

3 – Todas as heranças são completas e disjuntas. Completa, pois não existe outras subclasses. Disjunta, pois são classes exclusivas uma não pode sobrepor a outra.

**4 – Quando o objeto é mandado por parâmetro ou variável local, há economia de memória, pois assim que o método finaliza aquele objeto instanciado torna-se elegível para o Garbage Collector desalocar da memória, nota-se também que há aumento no encapsulamento, abaixando o acoplamento, porém, o desempenho cai.**

5 –

public class ControleCliente {

public void adicionarCliente(Cliente c) {

...

}

}

public class ControleCliente {

public void adicionarCliente() {

Cliente c = new Cliente();

...

}

}

6 – Foi escolhida a estrutura de List por ser um domínio que muitos dados podem ser repetidos. Não foi encontrado alguma relação de muitos que seria resolvida com a estrutura de Set.

7 – public class Cliente {

List<Animal> animal = new LinkedList<> ();

}

public class Animal {

List<Consulta> consulta = new LinkedList<> ();

}

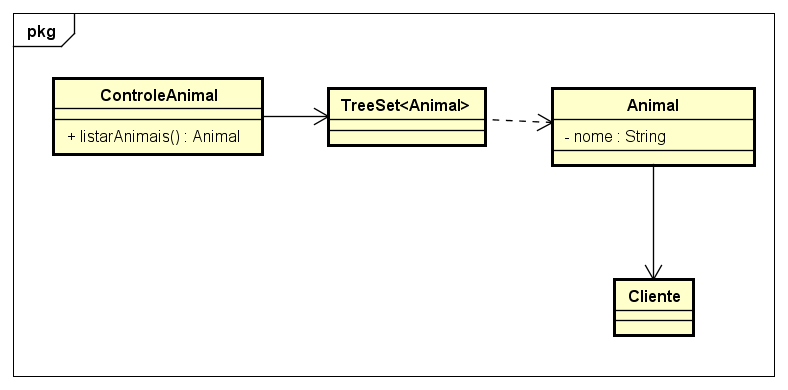
public class Historico {

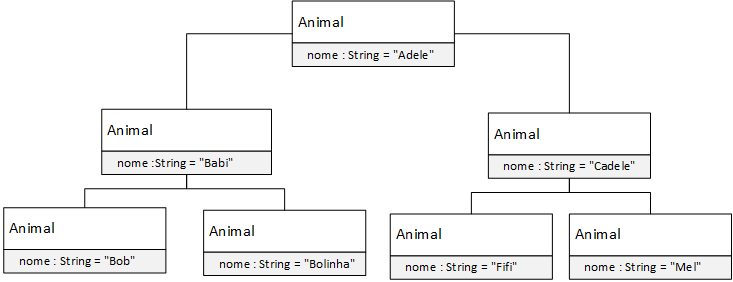
List<Exame> exame = new LinkedList<> ();

List<Consulta> consulta = new LinkedList<> ();

}

8 - A estrutura TreeSet compara os elementos e os ordena, os elementos são ordenados à medida que são adicionados na lista.





9 - public class Cliente {

TreeSet<Animal> animal = new TreeSet<Animal> ();

}

13 –

public class Pessoa {

public void ..(){

Veterinario = new Veterinario();

}

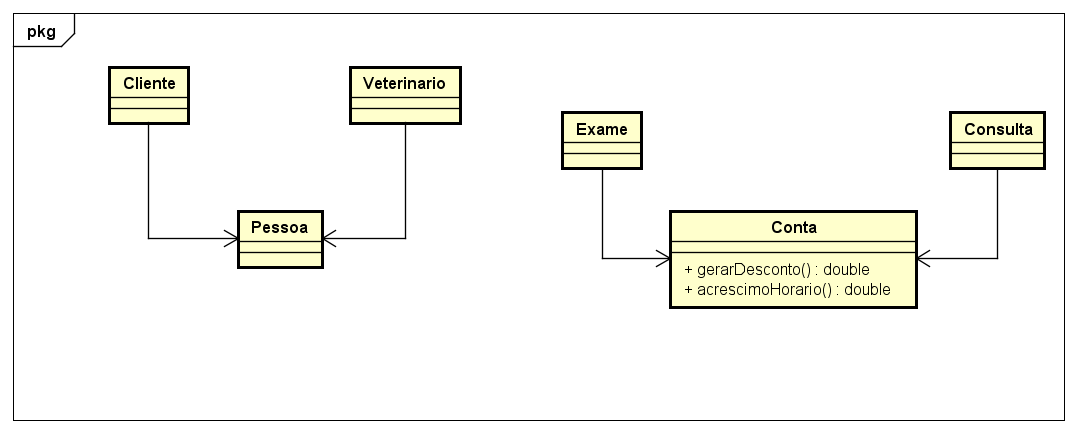
public void ..(){

Cliente = new Cliente();

}

}

14 – A instância do todo ficará na estrutura do Código das duas classes partes.



15 –

Public class Pessoa{

}

public class Veterinario{

Pessoa pessoa;

}

Public class Cliente{

Pessoa pessoa;  
}

Public class Conta{

}

Public class Exame{

Conta conta;

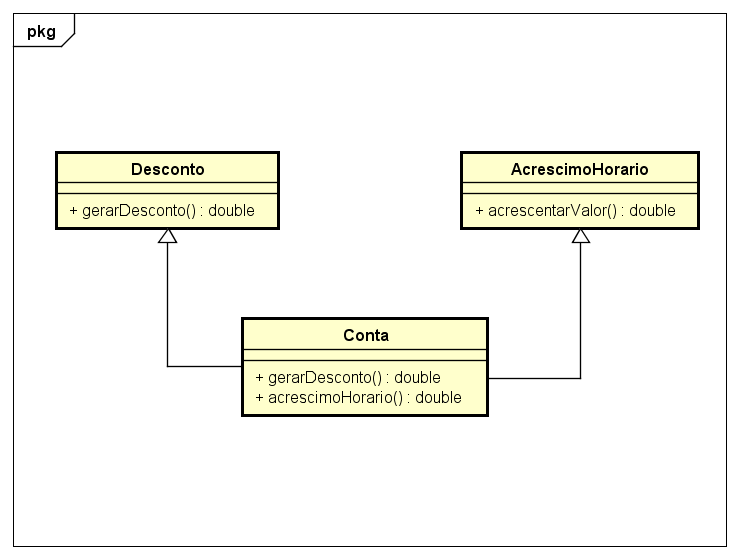
}

Public class Consulta{

Conta conta;

}

16 - A classe conta necessita de dois métodos que estão implementados em classes distintas.



17 - class desconto {

public:

double valorDesconto;

double gerarDesconto(double);

};

class acrescimoHorario {

public:

double valorAcrescimo;

double AcrescimoHorario(double);

};

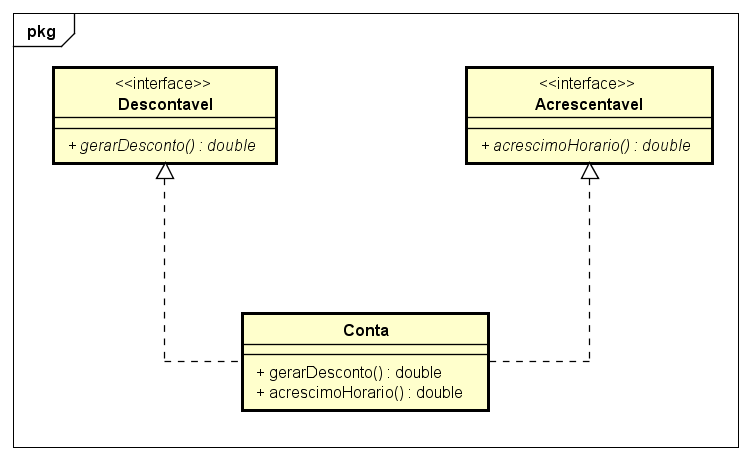
class conta : public desconto, public acrescimoHorario{

double gerarDesconto(double);

double AcrescimoHorario(double);

};

18 - As interfaces distintas consistem em métodos que serão utilizados pela a classe conta, sendo assim serão feitas duas realizações para assim a classe conta implementar o método concreto.



19-public interface Acrescentavel {

public double acrescimoHorario();

}

public interface Descontavel {

public double gerarDesconto();

}

public class Conta implements Descontavel, Acrescentavel {

private String horarioEmergencia;

private tipoPagamento formaPag;

private static double valorFinal;

private double valor;

private double valorDesconto;

private int valorAcrescimo;

@Override

public double gerarDesconto() {

…

}

@Override

public double acrescimoHorario() {

…

}

}

20 – A delegação possibilita contornar a limitação de algumas linguagens de programação no quesito de herança múltipla.

21-public class Conta{

AcrescimoHora=new AcrescimoHora();

private String horarioEmergencia;

private tipoPagamento formaPag;

private static double valorFinal;

private double valor;

private double valorDesconto;

private int valorAcrescimo;

public double gerarDesc () {

Desconto d = new Desconto();

d.gerarDesconto();

}

public double acrescimoHor () {

AcrescimoHorario ac=new AcrescimoHorario();

ac.acrescimoHora();

}

}

Public class AcrescimoHorario{

public double acrescimoHora () {

…

}

}

Public class Desconto{

public double gerarDesconto () {

…

}

}

22 - Faça um quadro comparativo entre reuso por generalização, realização e delegação, apresentando no mínimo duas vantagens e duas desvantagens para cada um desses conceitos.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Generalização | Realização | Delegação |
| Vantagem | Desempenho. | Manutenibilidade. | Alto encapsulamento. |
| Vantagem | Acoplamento forte. | Revelar as operações de um objeto sem revelar a sua classe. | Dinâmica. |
| Desvantagem | Diminuição do encapsulamento dependendo de sua profundidade. | Obrigação de ter que implementar todos os métodos. | Fraco acoplamento. |
| Desvantagem | Heranças múltiplas causam maior complexidade do sistema. | Dificuldade na reutilização do código devido aos métodos serem abstratos. | Baixo desempenho. |

23-

24-

25-