

1. 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.
2. Todas as heranças são completas e disjuntas. Completa, pois não existe outras subclasses. Disjunta, pois são classes exclusivas uma não pode sobrepor a outra.
3. 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 deslocar da memória, nota-se também que há aumento no encapsulamento, abaixando o acoplamento, porém, o desempenho cai.

public class ControleCliente {

public void adicionarCliente(Cliente c) {

...

}

}

public class ControleCliente {

public void adicionarCliente() {

Cliente c = new Cliente();

...

}

}

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

public class Cliente {

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

}

public class Animal {

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

}

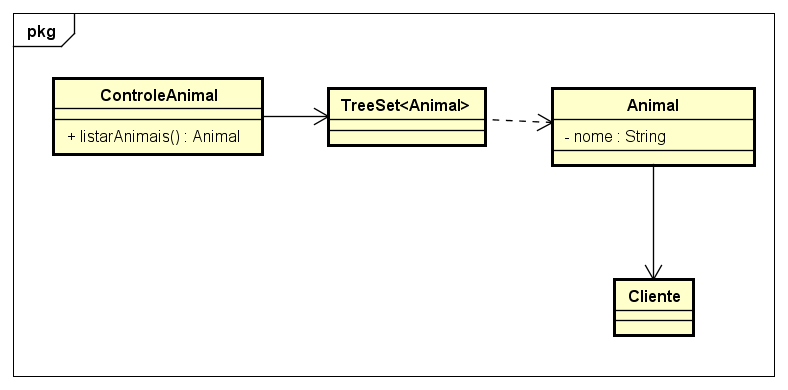
public class Historico {

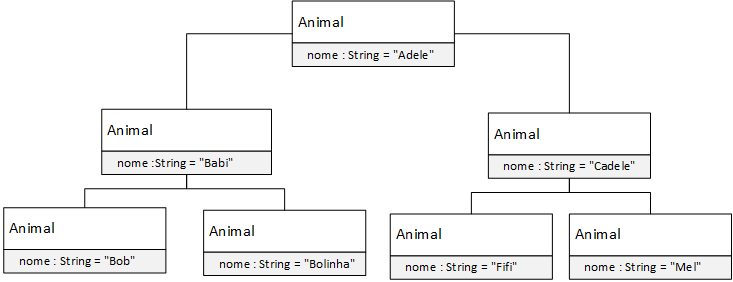
List<Exame> exame = new LinkedList<> ();

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

}

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

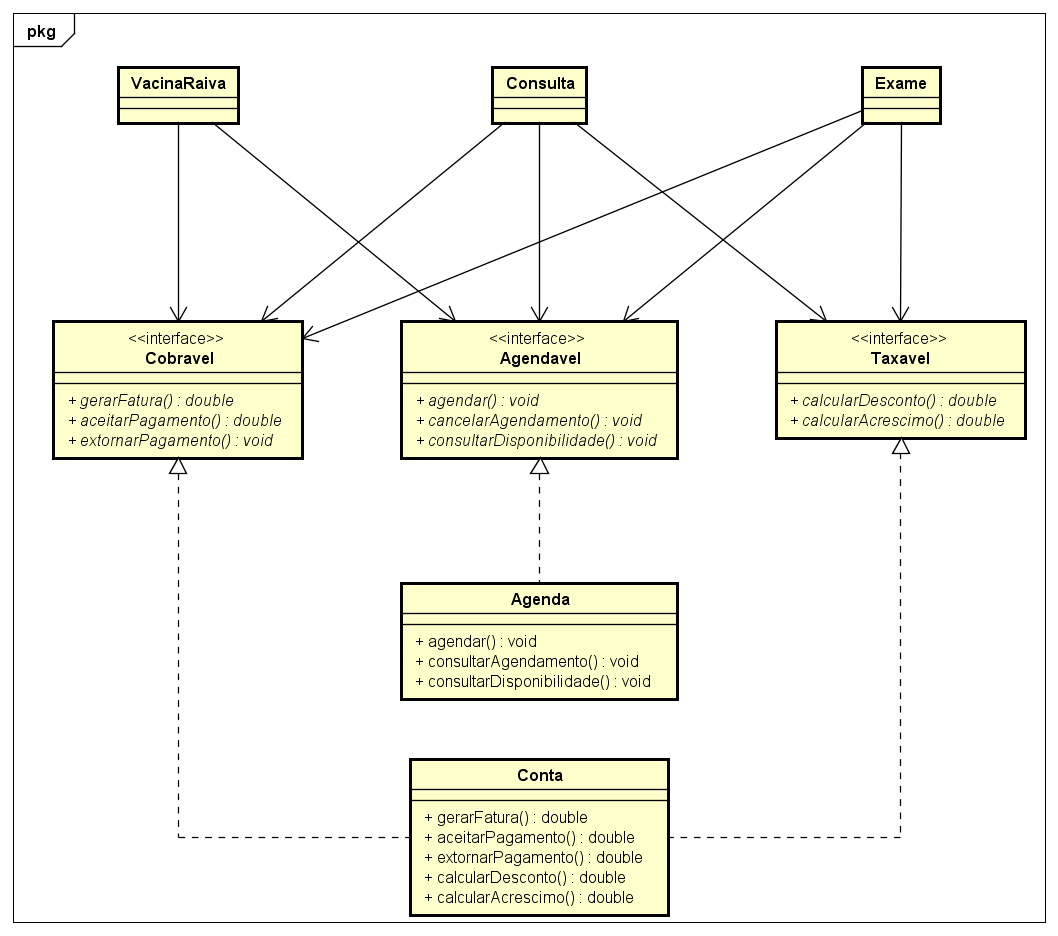




public class Cliente {

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

**10.**



**11.**

public interface Agendavel {

public void agendar();

public void cancelarAgendamento();

public void consultarDisponibilidade();

}

public interface Cobravel {

public double gerarFatura(double valor);

public double aceitarPagamento(double valor);

public void extornarPagamento();

}

public interface Taxavel {

public double calcularDesconto(double valor);

public double calcularAcrescimo(double valor);

}

public class Conta implements Cobravel, Taxavel {

@Override

public double calcularDesconto(double valor) {

return valor \* 0.9;

}

@Override

public double calcularAcrescimo(double valor) {

return valor \* 1.10;

}

@Override

public double gerarFatura(double valor) {

return valor;

}

@Override

public double aceitarPagamento(double valor) {

return valor;

}

@Override

public void extornarPagamento() {

...

}

}

public class Agenda implements Agendavel {

@Override

public void agendar() {

System.out.println("Consulta agendada");

}

@Override

public void cancelarAgendamento() {

System.out.println("Consulta cancelada");

}

@Override

public void consultarDisponibilidade() {

}

}

public class Consulta {

private Agendavel agendavel;

private Cobravel cobravel;

private Taxavel taxavel;

}

public class Exame {

private Agendavel agendavel;

private Cobravel cobravel;

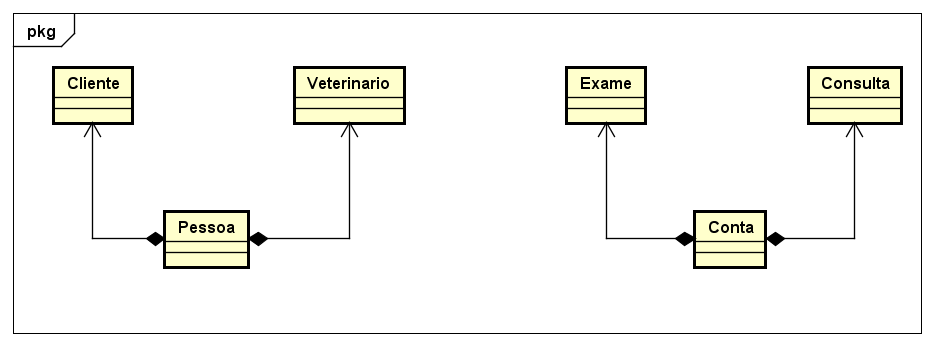
}

public class VacinaRaiva {

private Cobravel cobravel;

private Agendavel agendavel;

}

**12.** Uma pessoa pode ser veterinária e cliente ao mesmo tempo e conta pode ser conta de consulta e conta de exame ao mesmo tempo.

**13.**

public class Pessoa {

public void novoVeterinario(){

Veterinario v = new Veterinario();

...

}

public void novoCliente(){

Cliente = new Cliente();

...

}

}

public class Conta {

public void gerarContaExame() {

Exame e = new Exame();

...

}

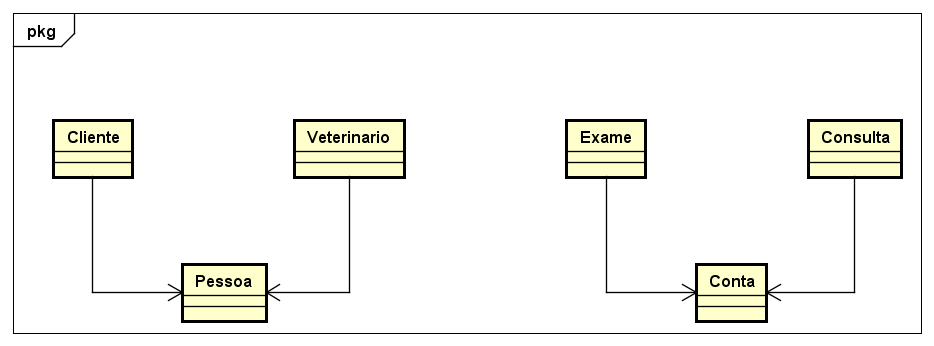
public void gerarContaConsulta () {

Consulta c = new Consulta();

...

}

**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;

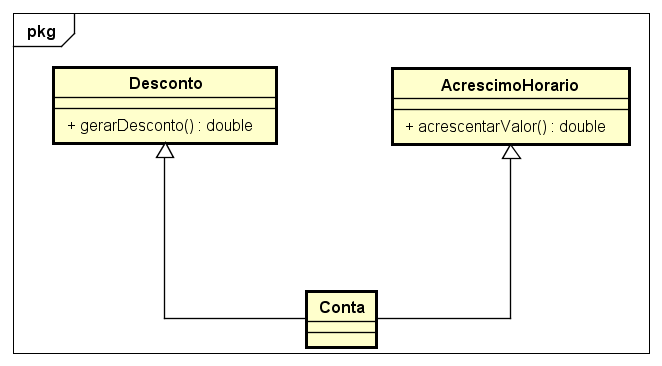
}

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);

};

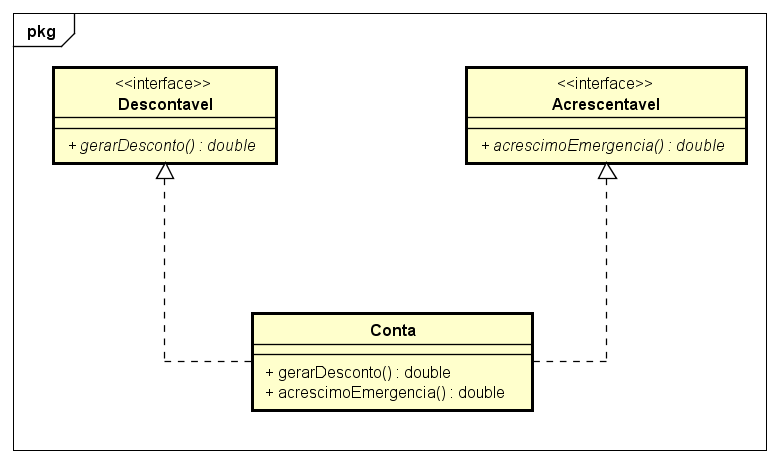
class conta : public Desconto, public AcrescimoHorario{

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 acrescimoEmergencia();

}

public interface Descontavel {

public double gerarDesconto();

}

public class Conta implements Descontavel, Acrescentavel {

private static double valorFinal;

private double valorDesconto;

private double valorAcrescimo;

@Override

public double gerarDesconto() {

…

}

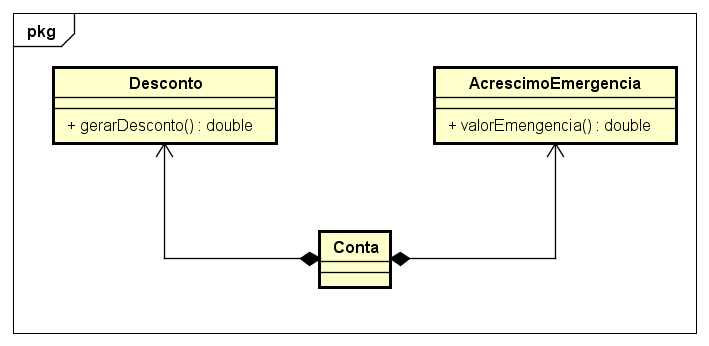
@Override

public double acrescimoHorario() {

…

}

}

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

**21.**

public class Conta {

private static double valorFinal;

private double valorDesconto;

private int valorAcrescimo;

public double gerarDesc () {

Desconto d = new Desconto();

valorDesconto = d.gerarDesconto();

}

public double acrescimoEmergencia () {

AcrescimoEmergencia ae = new AcrescimoEmergencia ();

valorAcrescimo = ae.valorEmergencia();

}

}

public class AcrescimoEmergencia {

public double acrescimoEmergencia () {

…

}

}

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.**