Engenharia de Software 3

Lista 2

Andrei Lucas Gonçalves 1680481521025

Daniel Augusto Scordamaglio 1680481521030

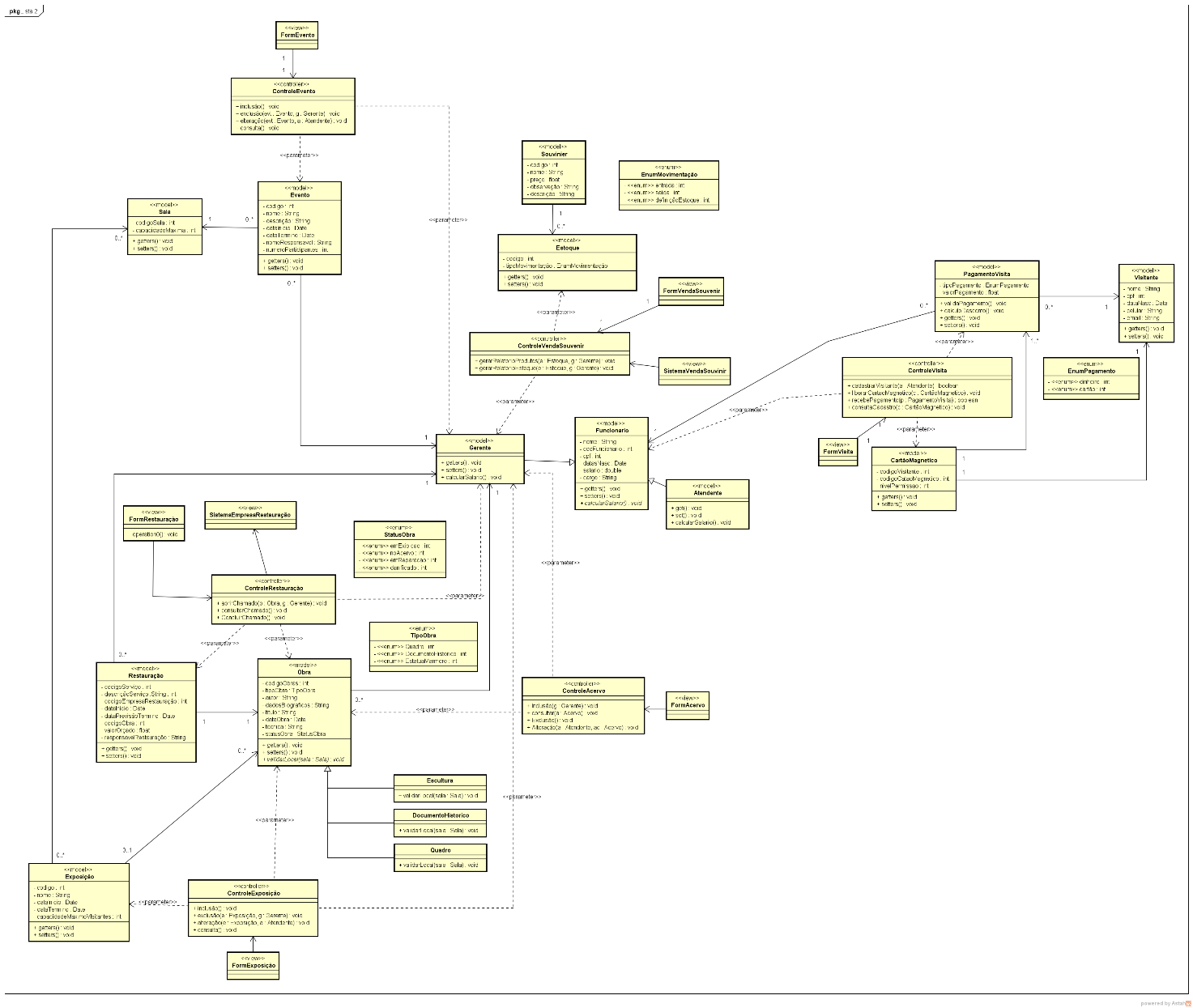
Diego de Melo Gonzaga 1680481521036

Elias Kyoharu Sanai 1680481521016

Gabriel Ferrai Dantas 1680481511015

Henrique Fernandes 141682263

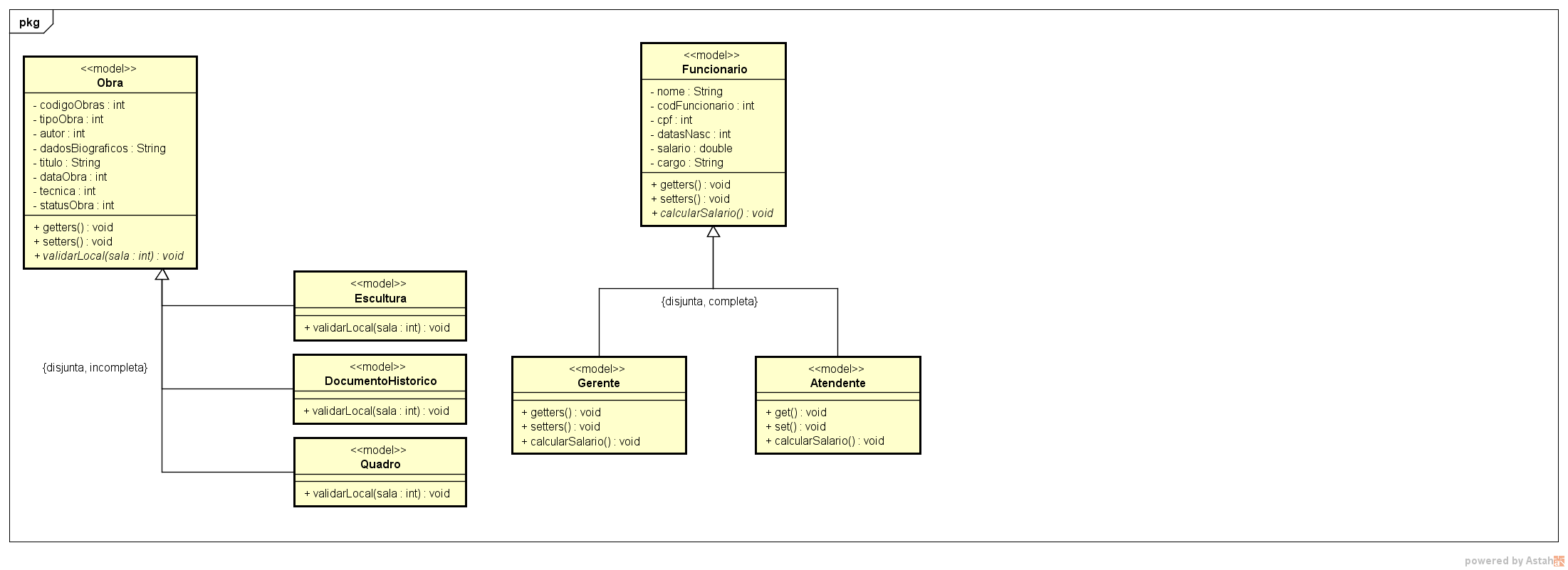
1. Construa uma nova versão refinada do diagrama de classes de projeto a partir dos exercícios 14, 17, 19, 21, 23, 25 e 27. Troque os <<estereótipos>> da categorização BCE pelos <<estereótipos>> do padrão de projeto MVC. A notação de classe deve continuar sendo mantida e as multiplicidades dos relacionamentos devem ser exibidas.



1. As relações de gen/espec modeladas apresentam classificação dinâmica? Justifique a tua resposta.

Não é necessário a classificação dinâmica, mas pode ser realizada na gen/espec de Gerente e Atendente, onde no caso de uso 1 o Gerente pode realizar a função de Atendente.

1. Quais restrições {OCL} sobre gen/espec são aplicáveis nas heranças modeladas? Justifique a tua resposta



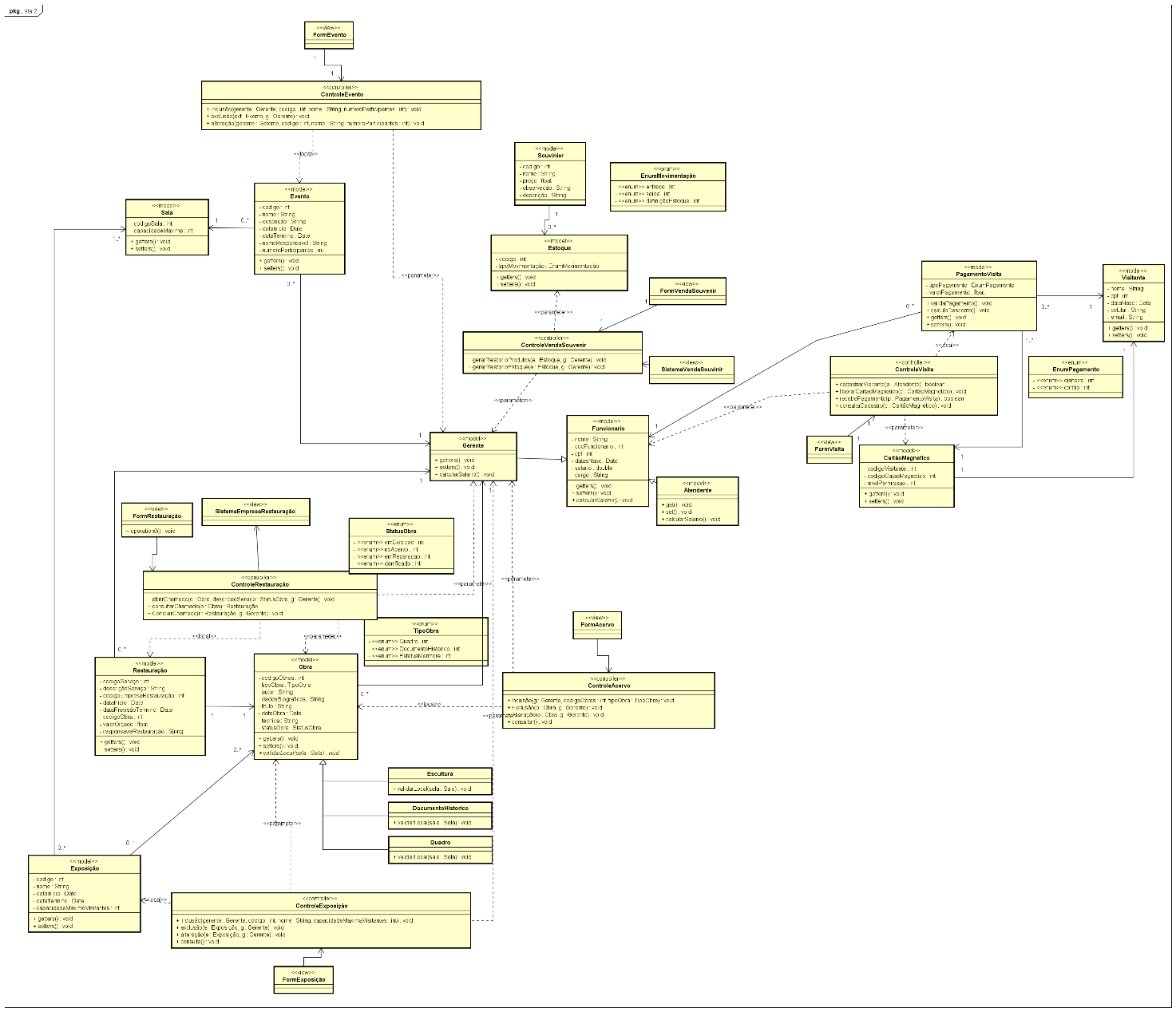
Heranças:

1. Obra, Escultura, Documento Histórico, Quadro {disjunta, incompleta}
2. Funcionario, Gerente, Atendente {disjunta, completa}

Na herança 1 de Obra ela é disjunta por não haver classificação dinâmica e incompleta porque ela poderá ser expandida (pode haver mais classes ainda não implementadas)

A herança 2 de Funcionario é disjunta também, é possível transformá-la em sobreposta, porém não é necessário, ela é completa pois, o domínio só exige o Atendente e o Gerente

1. Modele concomitantemente as dependências não estruturais por parâmetro e por variável local entre as classes de controle e modelo. Para cada relação entre as classes de controle e modelo, justifique o motivo de ter escolhido o tipo de dependência não estrutural.



ControleVisita:

para Gerente: Param, para validar o usuário

para CartãoMagnetico: Param, para usar um CartãoMagnetico já cadastrado

para PagamentoVisita: Local, Criação da Obra como variável local

ControleAcervo:

para Gerente: Param, para validar o usuário

para Obra: Local, Criação da Obra como variável local

ControleEvento:

para Gerente: Param, para validar o usuário

para Evento: Local, Criação do Evento como variável local

ControleRestauração:

para Gerente: Param, para validar o usuário

para Obra: Param, para usar uma obra já cadastrado

para Restauração: Local, Criação da Restauração como variável local

ControleExposição:

para Gerente: Param, para validar o usuário

para Obra: Param, para usar uma obra já criada

para Exposição: Local, Criação da Restauração como variável local

ControleVendaSouvenir:

para Gerente: Param, para validar o usuário

para Souvenir: Param, para usar um Souvenir já cadastrado

para Estoque: Param, para usar um Estoque já cadastrado

1. Apresente a estrutura básica de código em JAVA, C# ou C++ para implementar as dependências não estruturais por parâmetro e por variável local

class Restauracao {

//Atributos

//Propriedades (getters e setters)

}

class Obra {

//Atributos

//Propriedades (getters e setters)

}

class ControleRestauracao {

public void AbrirChamado(

Gerente gerente,

int codigoServico,

string descricaoServico,

//atributos da classe restauração...

) {

Restauracao restauracao = new Restauracao() {

CodigoServico = codigoServico,

DescricaoServico = descricaoServico,

//definição de outros atributos...

};

//Continuação do método (dependencia por variável local)

//nesse caso iria fazer algum uso de uma DAO por variável local também

}

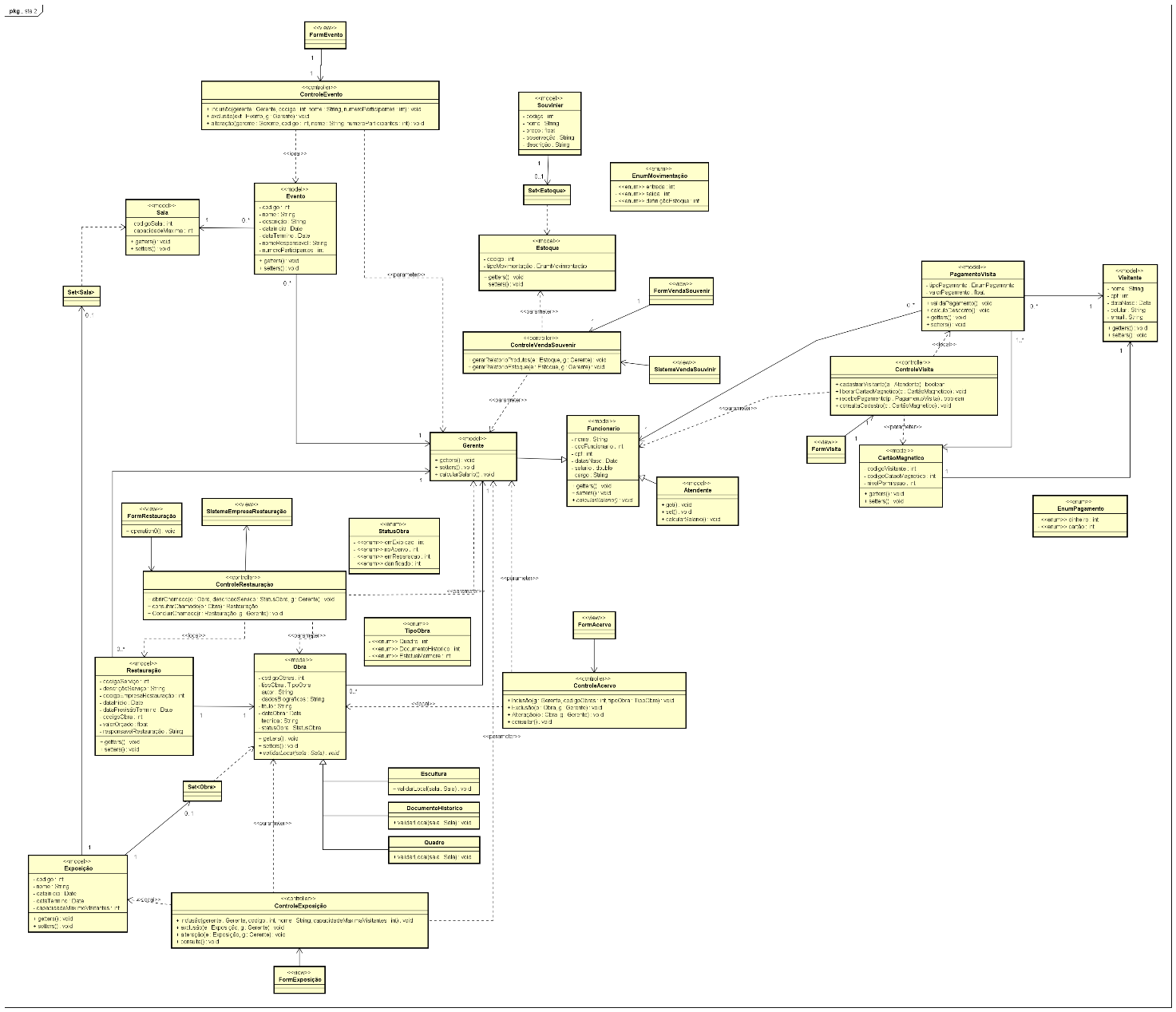
public Restauracao ConsultarChamado(Obra o) {

//Continuação do método (dependencia por parametro)

}

}

1. Modele concomitantemente as classes parametrizadas com a estrutura e para resolver o lado muitos dos relacionamentos. Para cada classe parametrizada modelada, justifique o motivo de ter escolhido o tipo de estrutura de dados.



Nesse diagrama só foram necessários 3 Set e nenhuma List e foram Set porque os elementos não irão se repetir Foram modificados:

1 Exposição para \* Sala

1 Exposição para \* Obra

1 Souvenir para \* Estoque

1. Apresente a estrutura básica de código em JAVA, C# ou C++ para implementar as classes parametrizadas com a estrutura <List> e <Set>.

class Exposicao {

private Set<Sala> salas = new HashSet<Sala>();

private Set<Obra> obras = new HashSet<Obra>();

//Outros Atributos...

public Set<Sala> Salas {

get { return salas; }

set { salas = value; }

}

public Set<Sala> Obras {

get { return obras; }

set { obras = value; }

}

//Outras Propriedades (getters e setters)...

}

class Souvenir {

private Set<Estoque> estoque = new HashSet<Estoque>();

//Outros Atributos...

public Set<Estoque> Estoque {

get { return estoque; }

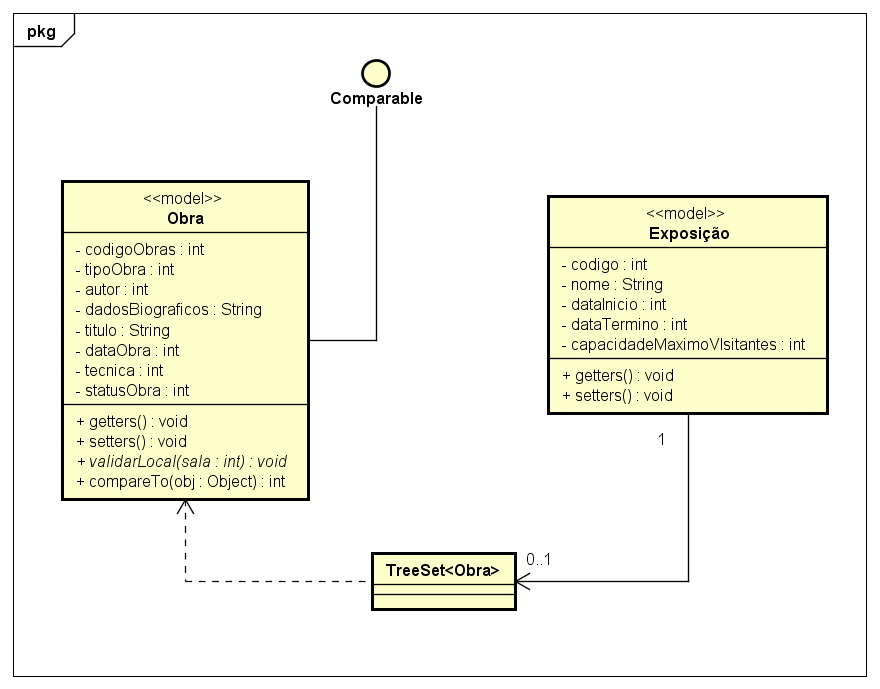
set { estoque = value; }

}

//Outras Propriedades (getters e setters)...

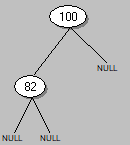
}

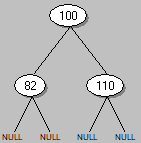
1. Modele a estrutura <TreeSet>. Justifique a razão dessa estrutura no seu diagrama. Represente graficamente como essa árvore trabalharia em tempo de execução.

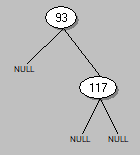


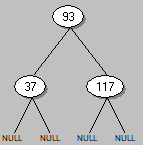
A entrada das Obras deve ser ordenada sem repetição podendo ordernar pelo codigoObra

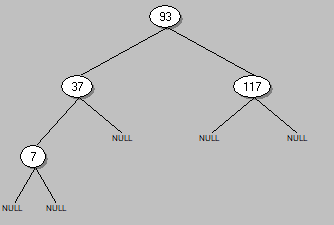
C:\Users\diego\AppData\Local\Microsoft\Windows\INetCache\Content.Word\01.png

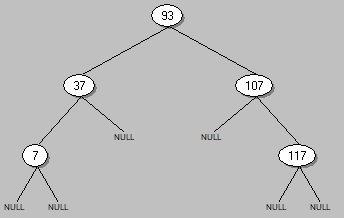


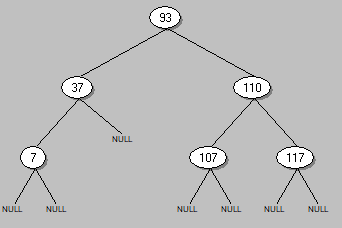


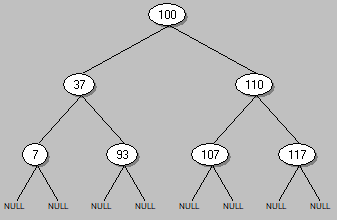


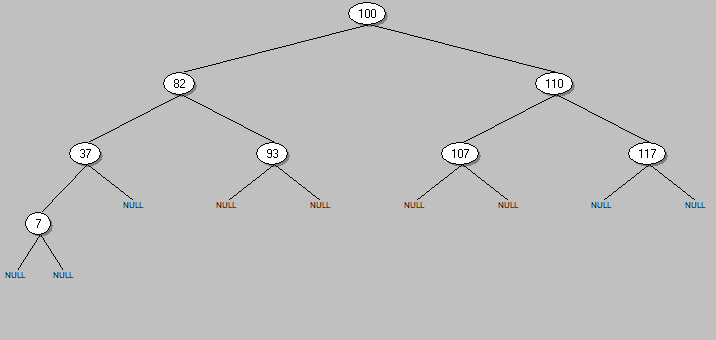


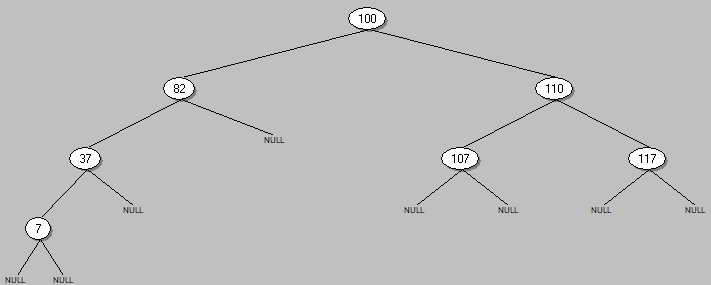


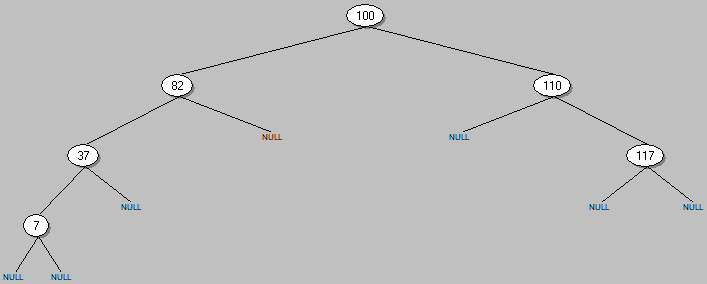


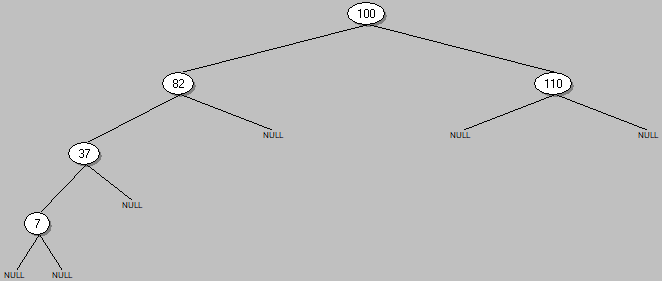


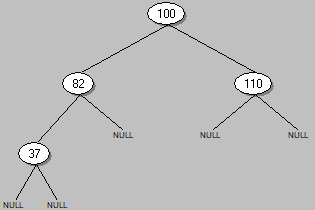












C:\Users\diego\AppData\Local\Microsoft\Windows\INetCache\Content.Word\15.png

1. Apresente a estrutura básica de código em JAVA, C# ou C++ para implementar a estrutura de dados <TreeSet>.

public class Obra implements Comparable {

private int codigoObras;

private int tipoObra;

private int autor;

private String dadosBiograficos;

private String titulo;

private int dataObra;

private int tecnica;

private int statusObra;

public void getters() {

//Código implementado em Java

}

public void setters() {

//Código implementado em Java

}

public abstract void validarLocal(int sala);

/\*\*

\* @see Comparable#compareTo(obj:Object)()

\*/

public int compareTo(obj:Object)() {

//Código implementado em Java

}

}

public class JavaApplication {

public static void main(String[] args) {

// TODO code application logic here

TreeSet<CartaoMagnetico> lista = new TreeSet();

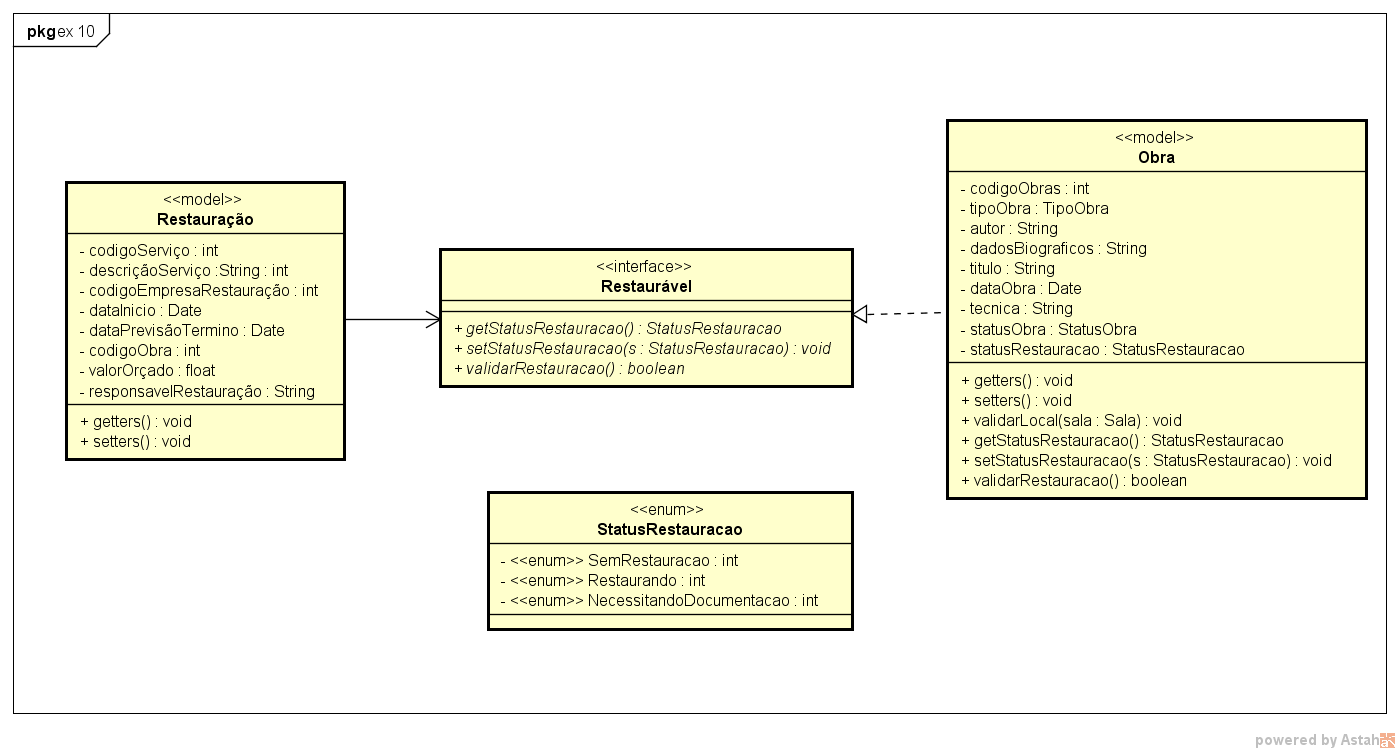
lista.add(new Obra());

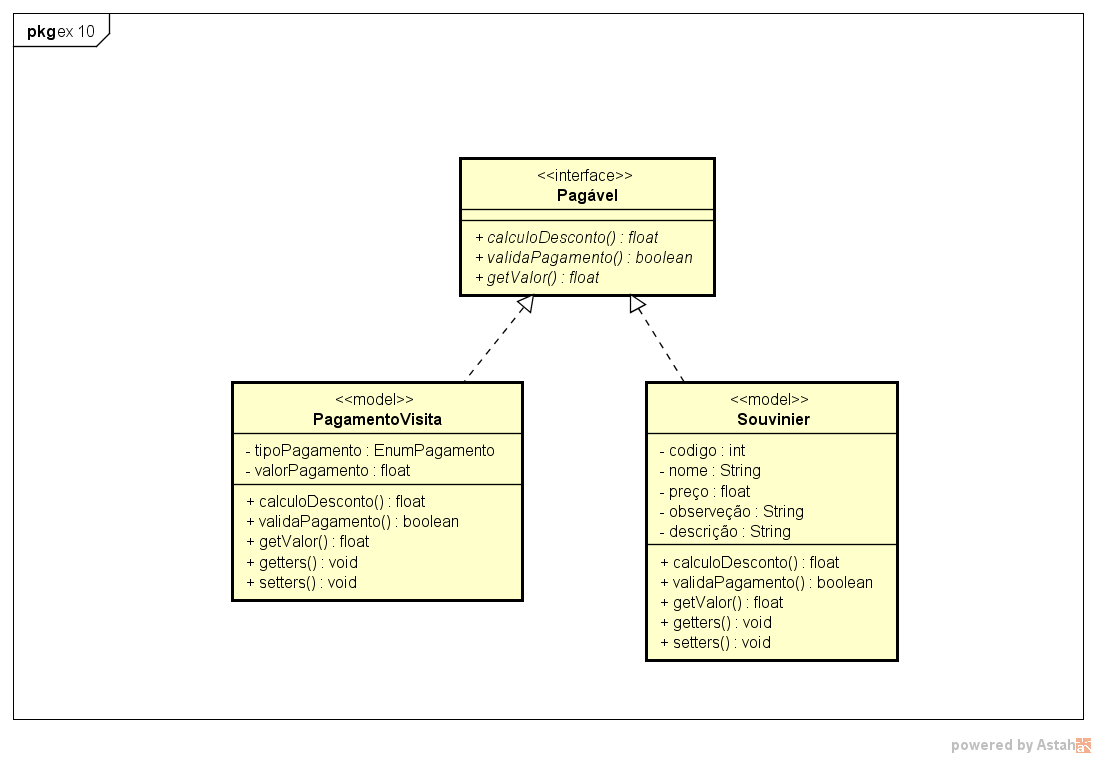
lista.add(new Obra());

}

}

1. Modele duas relações de interface com nomes adjetiváveis, estabelecendo o devido contrato de comportamento entre as classes consumidoras e fornecedoras e declarando as operações nas interfaces a serem implementadas pelas classes fornecedoras. Justifique a razão de existência de cada uma das relações de interface.





Interface Restaurável

Por meio dessa interface que protege de outras informações desnecessárias(aumentando o encapsulamento) para restauração as classes consumidoras(nesse caso somente a Restauração) podem coletar o status de restauracao e validar a restauração

Interface Pagável

Desta forma as classes que forem consumir a interface Pagável e ultilizar seus métodos (calculoDesconto(), validaPagamento(), getValor()) não acessam outras informações (aumentando o encapsulamento)

1. Apresente a estrutura básica de código em JAVA ou C# para implementar as relações de interface.

//INTERFACE 1 - Restauravel

public enum StatusRestauracao {

SemRestauracao,

Restaurando,

NecessitandoDocumentacao

}

public interface Restaurável {

StatusRestauracao getStatusRestauracao();

void setStatusRestauracao(StatusRestauracao s);

boolean validarRestauracao();

}

public class Restauração {

private Restaurável restaurável;

//Outros atributos e propriedades

}

public class Obra : Restaurável {

private StatusRestauracao statusRestauracao;

//Outros atributos e propriedades

public StatusRestauracao getStatusRestauracao() {

return statusRestauracao;

}

public void setStatusRestauracao(StatusRestauracao s) {

statusRestauracao = s;

}

public boolean validarRestauracao() {

//Aqui faria a validação da restauração

}

}

//INTERFACE 2 - Pagavel

public class Souvinier : Pagável {

//Outros atributos e propriedades

public float calculoDesconto() {

//Realiza o calculo do desconto pela classe Souvenir

}

public boolean validaPagamento() {

//Valida o pagamento pela classe Souvenir

}

public float getValor() {

//Retorna o preço do Souvenir

}

}

public interface Pagável {

float calculoDesconto();

boolean validaPagamento();

float getValor();

}

public class PagamentoVisita : Pagável {

//Outros atributos e propriedades

public float calculoDesconto() {

//Realiza o calculo do desconto pela classe PagamentoVisita

}

public boolean validaPagamento() {

//Valida o pagamento pela classe PagamentoVisita

}

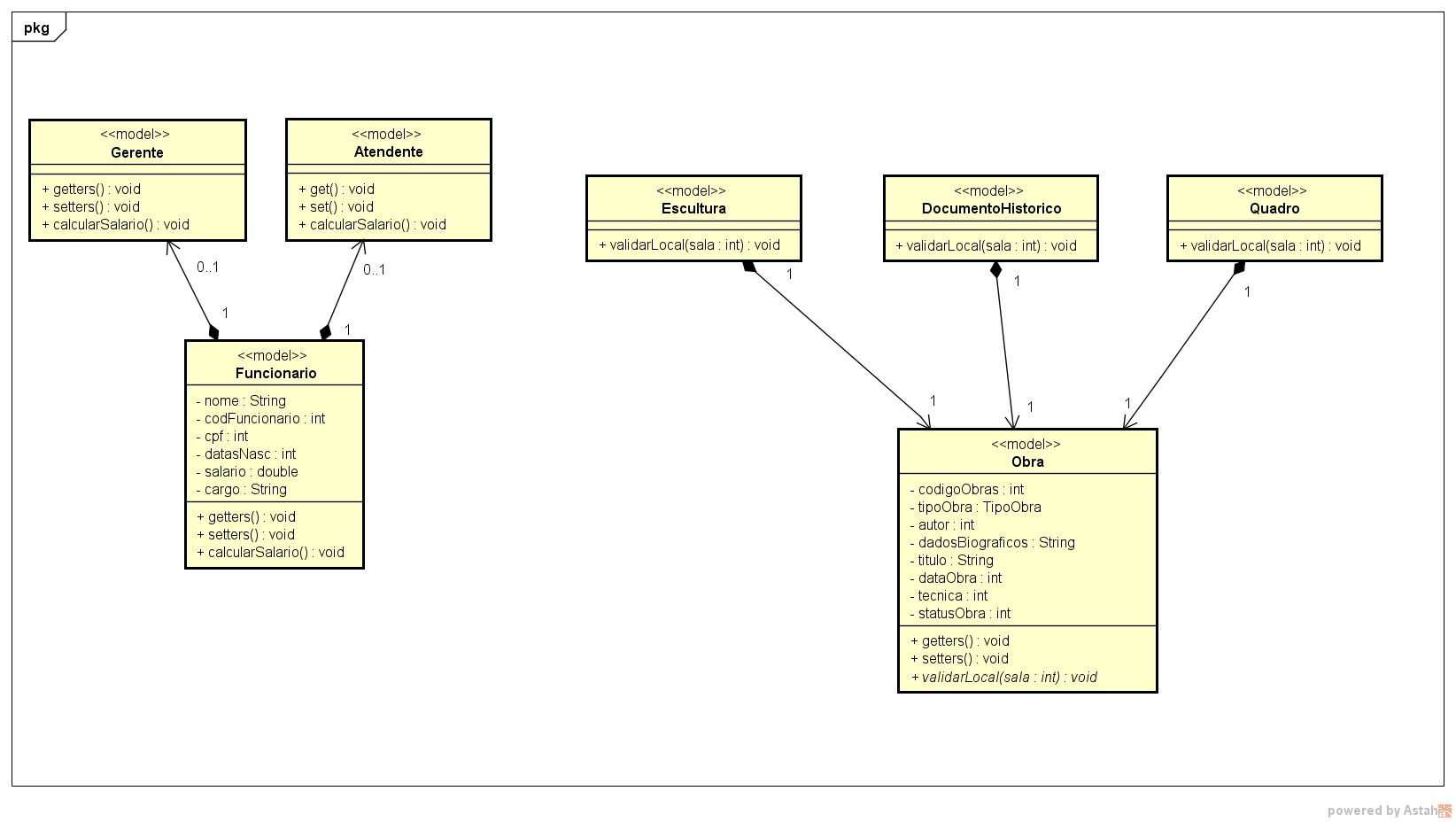
public float getValor() {

//Retorna o preço do PagamentoVisita

}

}

1. - Modele duas relações de delegação com relacionamento de composição, utilizando classes diferentes para cada uma. Justifique a razão de existência de cada uma das relações de delegação.



No exemplo demonstrado a relação de Gerente, Atendente e Funcionario acaba permitindo classificação dinamica por meio da delegação, nesse caso é válido já que às vezes o Gerente precisa realizar o cargo de Atendente no sistema.

No exemplo demonstrado em relação a Obra é válido, porém diferente do caso do Funcionario pois dessa vez a delegação está invertida, as classes filhas usaram a mãe que é a Obra, e na classe Obra que usará somente seus atributos podendo realizar o inicio do método validarlocal() genérico também

1. Apresente a estrutura básica de código em JAVA, C# ou C++ para implementar as relações de delegação com relacionamento de composição.

//CLASSES GERENTE ATENDENTE E FUNCIONARIO

class Gerente {

//Outros atributos e propriedades

public float CalcularSalario() {

//Calculo pelo Gerente

}

}

class Atendente {

//Outros atributos e propriedades

public float CalcularSalario() {

//Calculo pelo Atendente

}

}

class Funcionario {

private Gerente gerente;

private Atendente atendente;

//Outros atributos e propriedades

public void DefinirGerente() {

gerente = new Gerente();

}

public void DefinirAtendente() {

atendente = new Atendente();

}

public float CalcularSalarioPorAtendente() {

return atendente.CalcularSalario();

}

public float CalcularSalarioPorGerente() {

return gerente.CalcularSalario();

}

}

//CLASSES OBRAS

class Obra {

//Atributos e propriedades

}

class Escultura {

private Obra obra;

//Outros atributos e propriedades

public Escultura() {

obra = new Obra();

}

}

class DocumentoHistorico {

private Obra obra;

//Outros atributos e propriedades

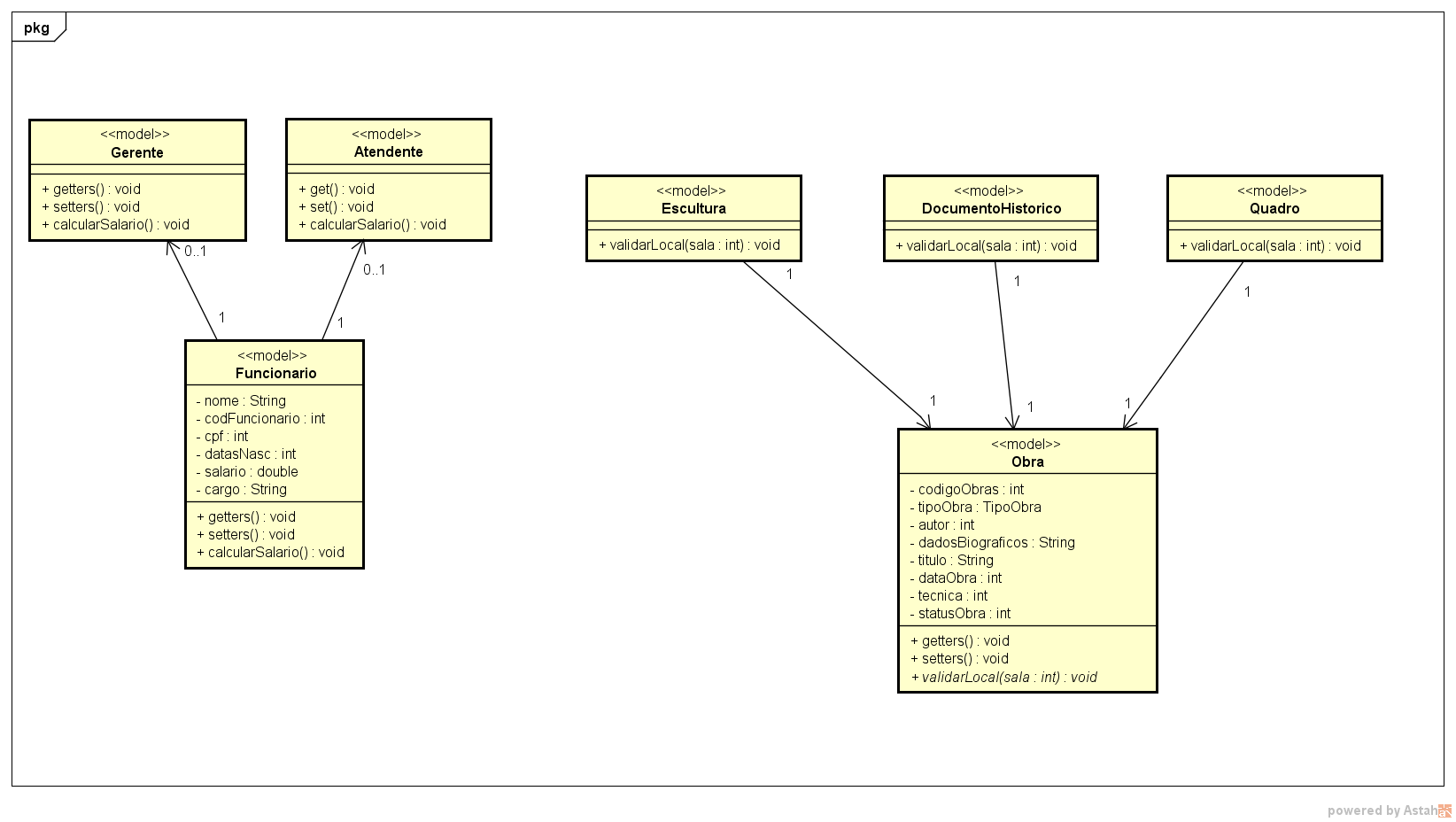
public DocumentoHistorico() {

obra = new Obra();

}

}

1. Transforme as duas relações de delegação com relacionamento de composição para relações de delegação com dependência estrutural. Justifique a tua resposta.



Praticamente o mesmo do que no exercício 12, porém com a diferença de que caso o Funcionario seja destruido, os objetos Gerente e Atendente poderão continuar em memória, no caso de Obra se aplica o mesmo também

1. Apresente a estrutura básica de código em JAVA, C# ou C++ para implementar as relações de delegação com relacionamento de dependência estrutural.

//CLASSES GERENTE ATENDENTE E FUNCIONARIO

class Gerente {

//Outros atributos e propriedades

public float CalcularSalario() {

//Calculo pelo Gerente

}

}

class Atendente {

//Outros atributos e propriedades

public float CalcularSalario() {

//Calculo pelo Atendente

}

}

//

// O que mudou aqui é que o Gerente e Atendente podem ser instanciados

// por meio de propriedades (setters)

//

class Funcionario {

private Gerente gerente;

public Gerente Gerente {

get { return gerente; }

set { gerente = value; }

}

private Atendente atendente;

public Atendente Atendente {

get { return atendente; }

set { atendente = value; }

}

//Outros atributos e propriedades

public float CalcularSalarioPorAtendente() {

return atendente.CalcularSalario();

}

public float CalcularSalarioPorGerente() {

return gerente.CalcularSalario();

}

}

//CLASSES OBRAS

//

// O que mudou aqui é que a obra agora pode ser enviado por

// meio de propriedades

//

class Obra {

//Atributos e propriedades

}

class Escultura {

private Obra obra;

public Obra Obra {

get { return atendente; }

set

{

if(value == null)

throw new NullReferenceException();

obra = value;

}

}

//Outros atributos e propriedades

}

class DocumentoHistorico {

private Obra obra;

public Obra Obra {

get { return atendente; }

set

{

if(value == null)

throw new NullReferenceException();

obra = value;

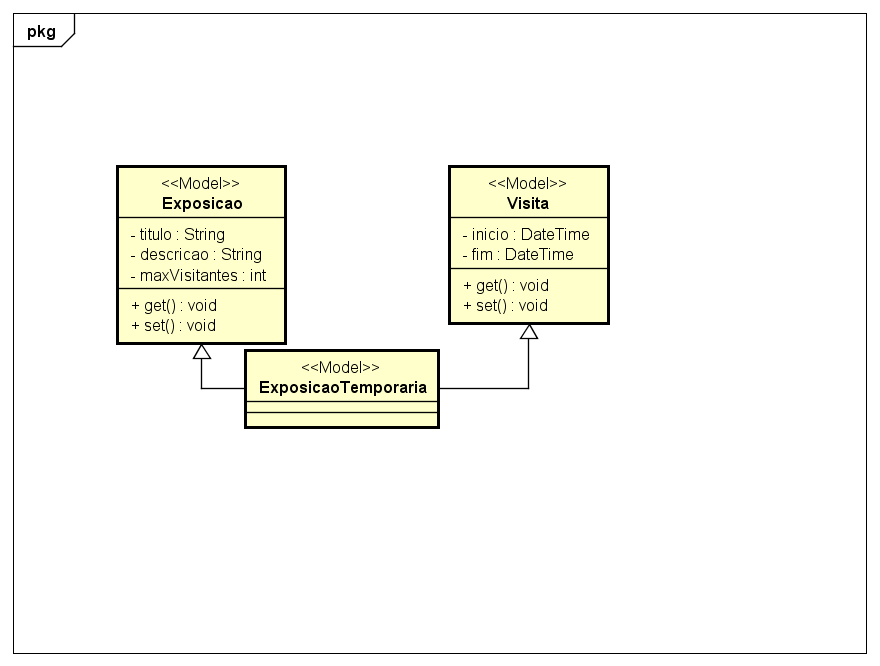
}

}

//Outros atributos e propriedades

}

1. Independentemente das relações de gen/espec já existentes no diagrama, abstraia o domínio e modele uma relação de herança múltipla. Justifique a razão de existência dessa herança múltipla.



Conforme existe exposições que são fixas, existem aquelas exposições que passam determinado tempo e depois vão embora como uma Visita.

Utilizar generalização aumenta o desempenho do programa.

1. Apresente a estrutura básica de código em C++ para implementar a relação de herança múltipla.

class Visita

{

private:

DateTime inicio;

DateTime fim;

public:

void get();

void set();

};

class Exposicao

{

private:

String titulo;

String descricao;

int maxVisitantes;

public:

void get();

void set();

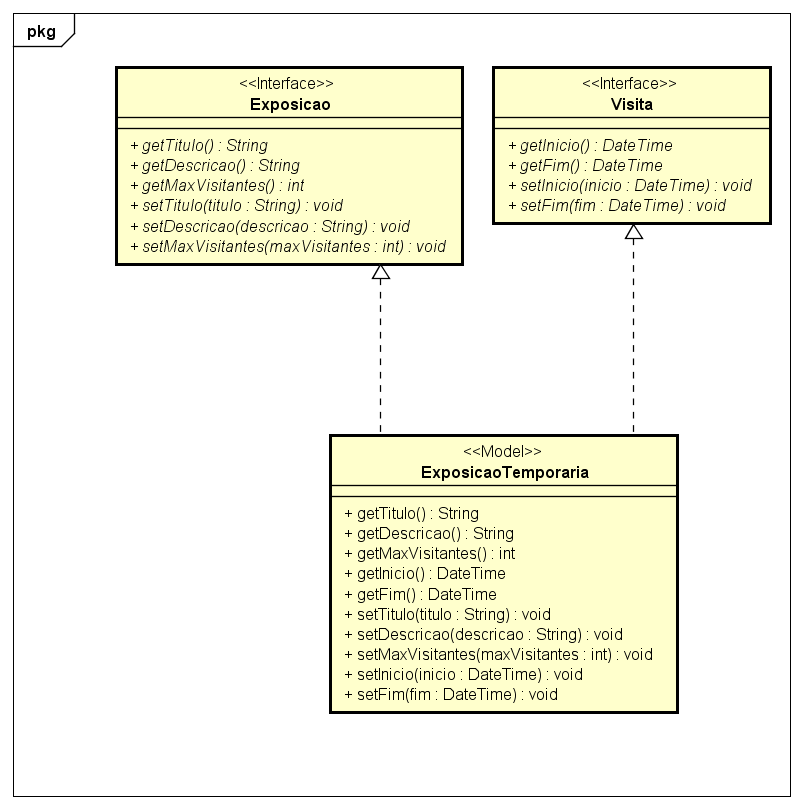
};

class ExposicaoTemporaria : public Exposicao, public Visita

{

};

1. Transforme a relação de herança múltipla para uma relação de realização. Justifique a tua resposta.



Seguindo o princípio SOLID é melhor manter diversas interfaces específicas do que uma geral.

Além disso utilizar interfaces é uma maneira muito boa de manter o Principio Orientado a Objeto, pois interfaces encapsulam os mesmos.

1. Apresente a estrutura básica de código em JAVA ou C# para implementar a relação de realização

public interface Exposicao {

public abstract String getTitulo();

public abstract String getDescricao();

public abstract int getMaxVisitantes();

public abstract void setTitulo(String titulo);

public abstract void setDescricao(String descricao);

public abstract void setMaxVisitantes(int maxVisitantes);

}

public interface Visita {

public abstract DateTime getInicio();

public abstract DateTime getFim();

public abstract void setInicio(DateTime inicio);

public abstract void setFim(DateTime fim);

}

public class ExposicaoTemporaria implements Exposicao, Visita {

public String getTitulo() {

return null;

}

public String getDescricao() {

return null;

}

public int getMaxVisitantes() {

return 0;

}

public DateTime getInicio() {

return null;

}

public DateTime getFim() {

return null;

}

public void setTitulo(String titulo) {

}

public void setDescricao(String descricao) {

}

public void setMaxVisitantes(int maxVisitantes) {

}

public void setInicio(DateTime inicio) {

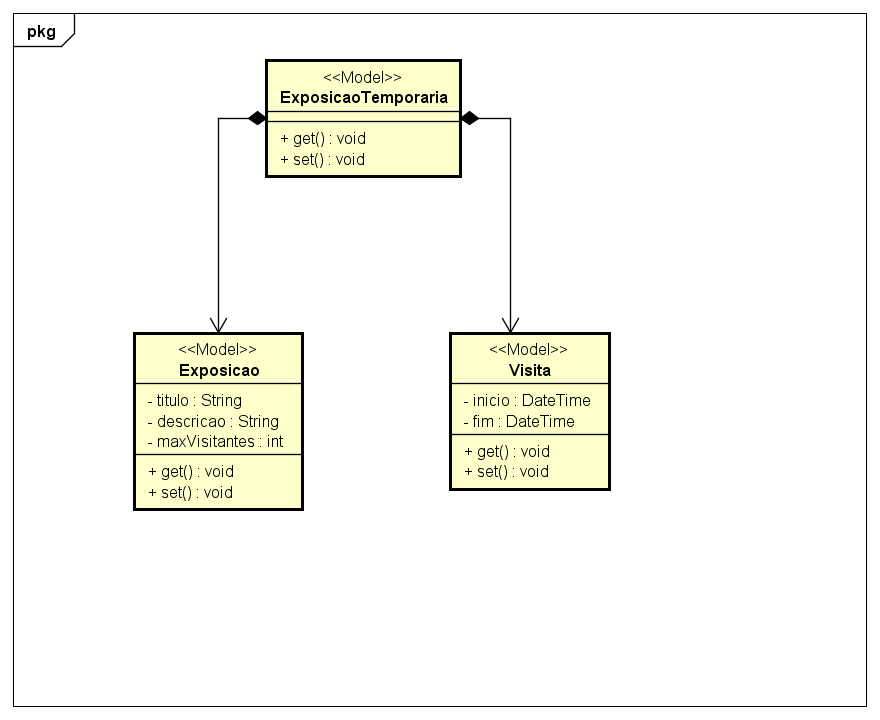
}

public void setFim(DateTime fim) {

}

}

1. Transforme a relação de herança múltipla para uma relação de delegação com relacionamento de composição. Justifique a tua resposta.



Desta forma aumenta o dinamismo do objeto e somente precisa criar os objetos Visita e/ou Obra quando forem necessário.

1. - Apresente a estrutura básica de código em JAVA, C# ou C++ para implementar a relação de delegação com relacionamento de composição.

public class Exposicao {

private String titulo;

private String descricao;

private int maxVisitantes;

public void get() {

}

public void set() {

}

}

public class Visita {

private DateTime inicio;

private DateTime fim;

public void get() {

}

public void set() {

}

}

public class ExposicaoTemporaria {

private Exposicao exposicao = null;

private Visita visita = null;

public void get() {

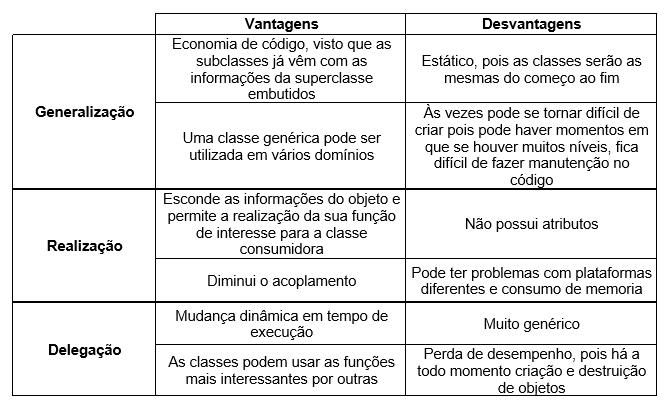
}

public void set() {

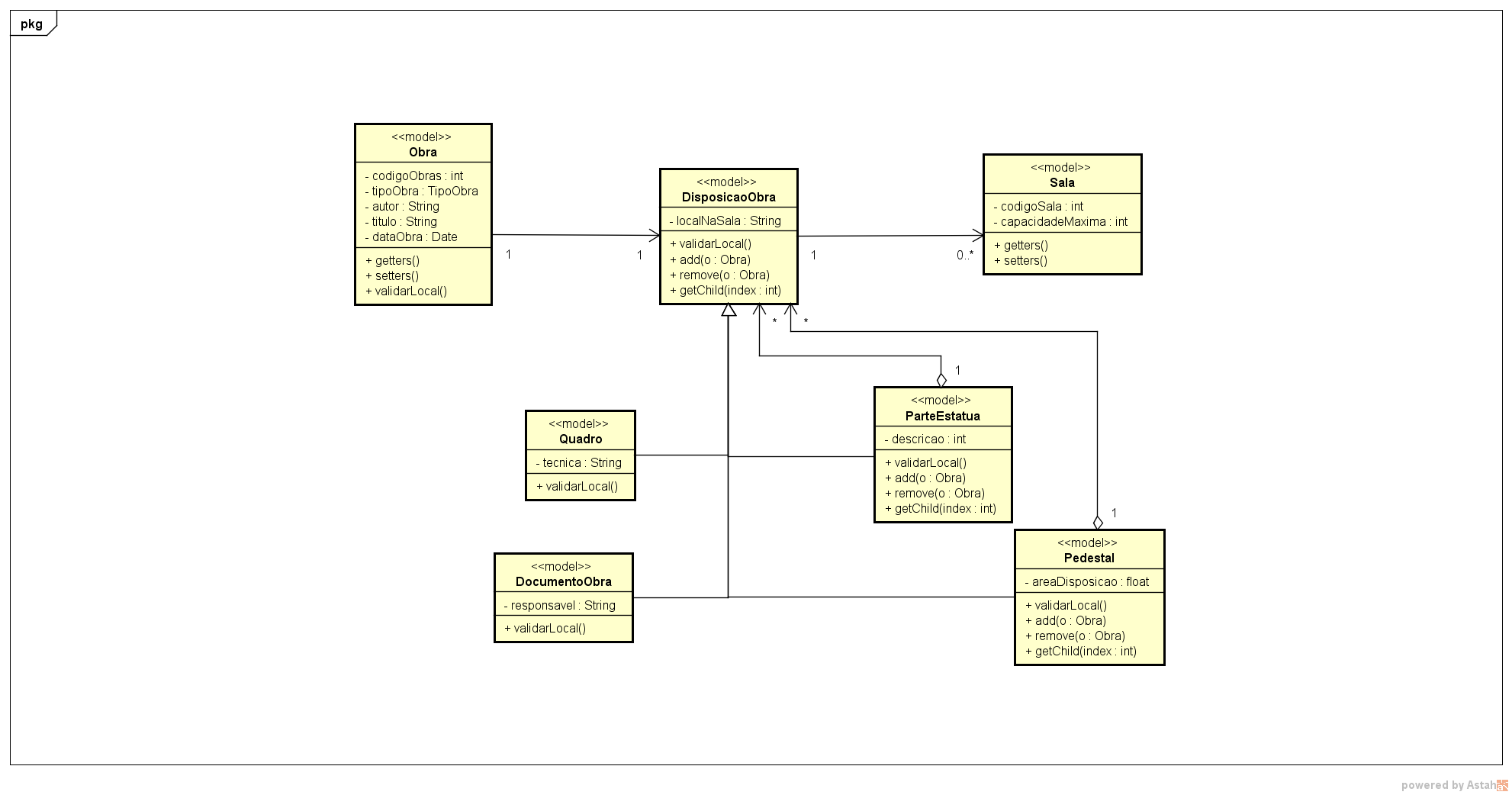
}

}

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



1. Modele o padrão de projeto Composite. Qual o propósito desse padrão no diagrama de classes de projeto?



A disposição de uma Obra de arte na sala pode ser feita com muitos elementos, especialmente para Obras de artes complexas que usam muitos materiais como por exemplo pedestais, vários espaços como Múltiplos quadros que juntos formam uma Obra de arte, isso justificaria o uso do Composite que por meio de recursividade acessa a disposição da Obra na sala usando a classe DisposicaoObra

1. - Apresente a estrutura básica de código em JAVA, C# ou C++ para implementar o padrão de projeto Composite.

class Obra {

//Atributos e Propriedades

}

//Component do Composite

class DisposicaoObra {

protected List<DisposicaoObra> listDisposicao = new List<DisposicaoObra>();

private string local;

public string Local {

get { return local; }

set { local = value; }

}

public virtual void Add(DisposicaoObra d) {

listDisposicao.Add(d);

}

public virtual void Remove(DisposicaoObra d) {

listDisposicao.Remove(d);

}

public virtual void ValidarLocal() {

//Valida das filhas

foreach (var item in listDisposicao)

{

item.ValidarLocal();

}

//Valida o local atual desse componente

}

}

//A Folha do Composite

class Quadro : DisposicaoObra {

public override void ValidarLocal() {

//Valida o local atual desse componente

}

}

//Um composite

class Pedestal : DisposicaoObra {

public override void ValidarLocal() {

//Valida das filhas

foreach (var item in listDisposicao)

{

item.ValidarLocal();

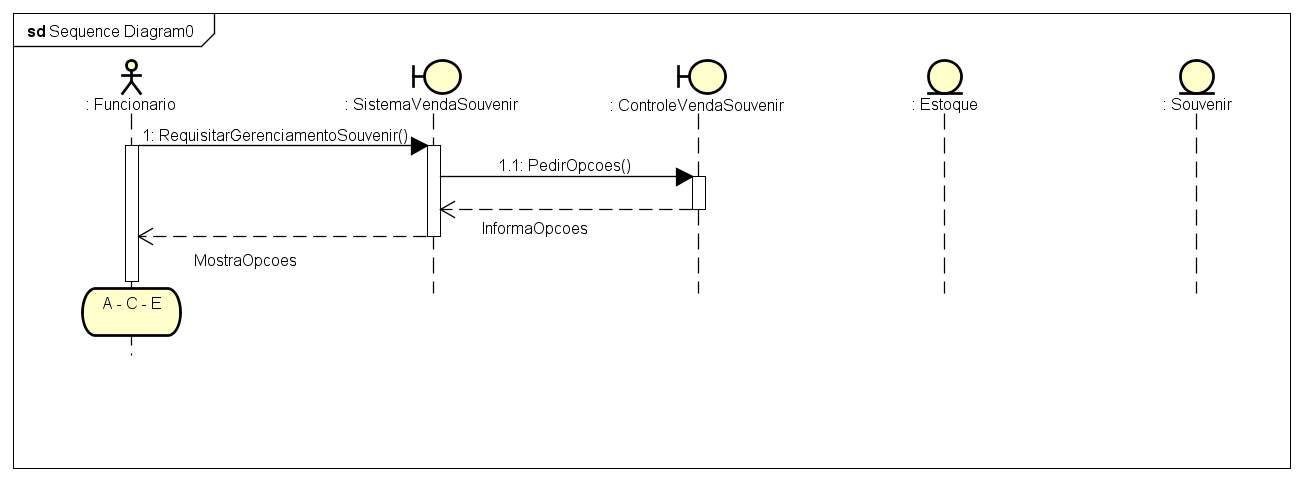
}

//Valida o local atual desse componente

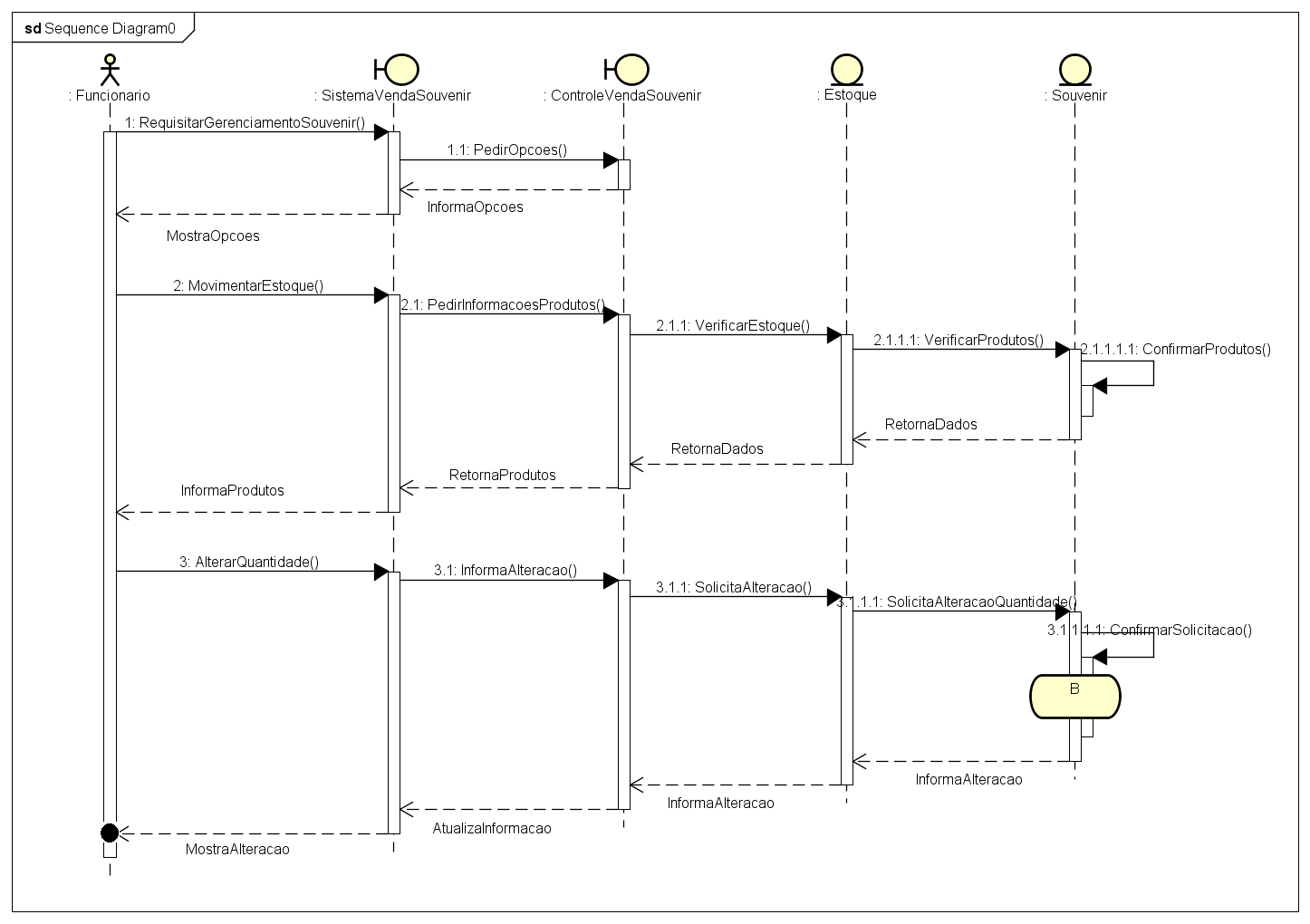
}

}

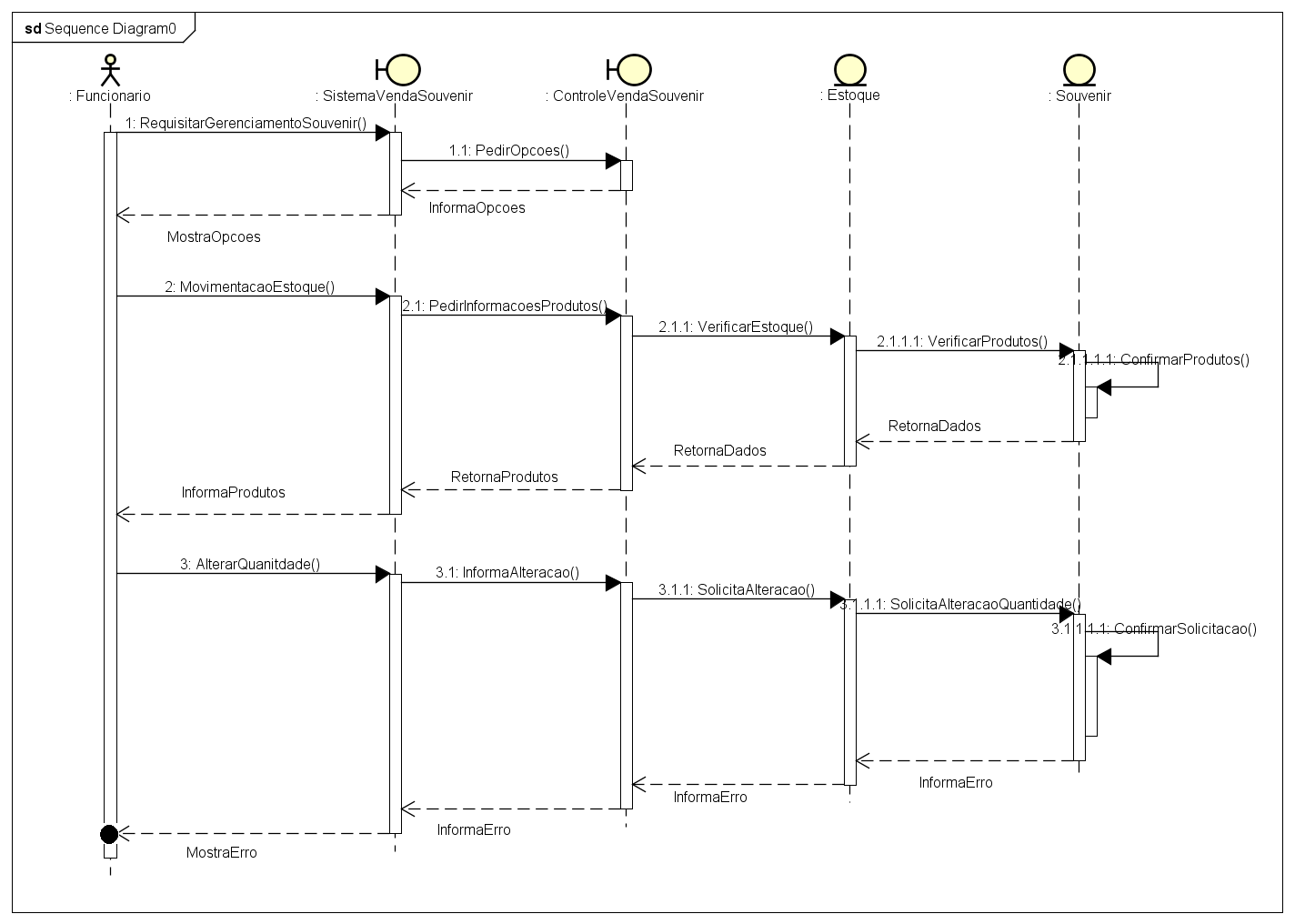
1. Construa um diagrama de sequência para o CSU06 utilizando notação MVC



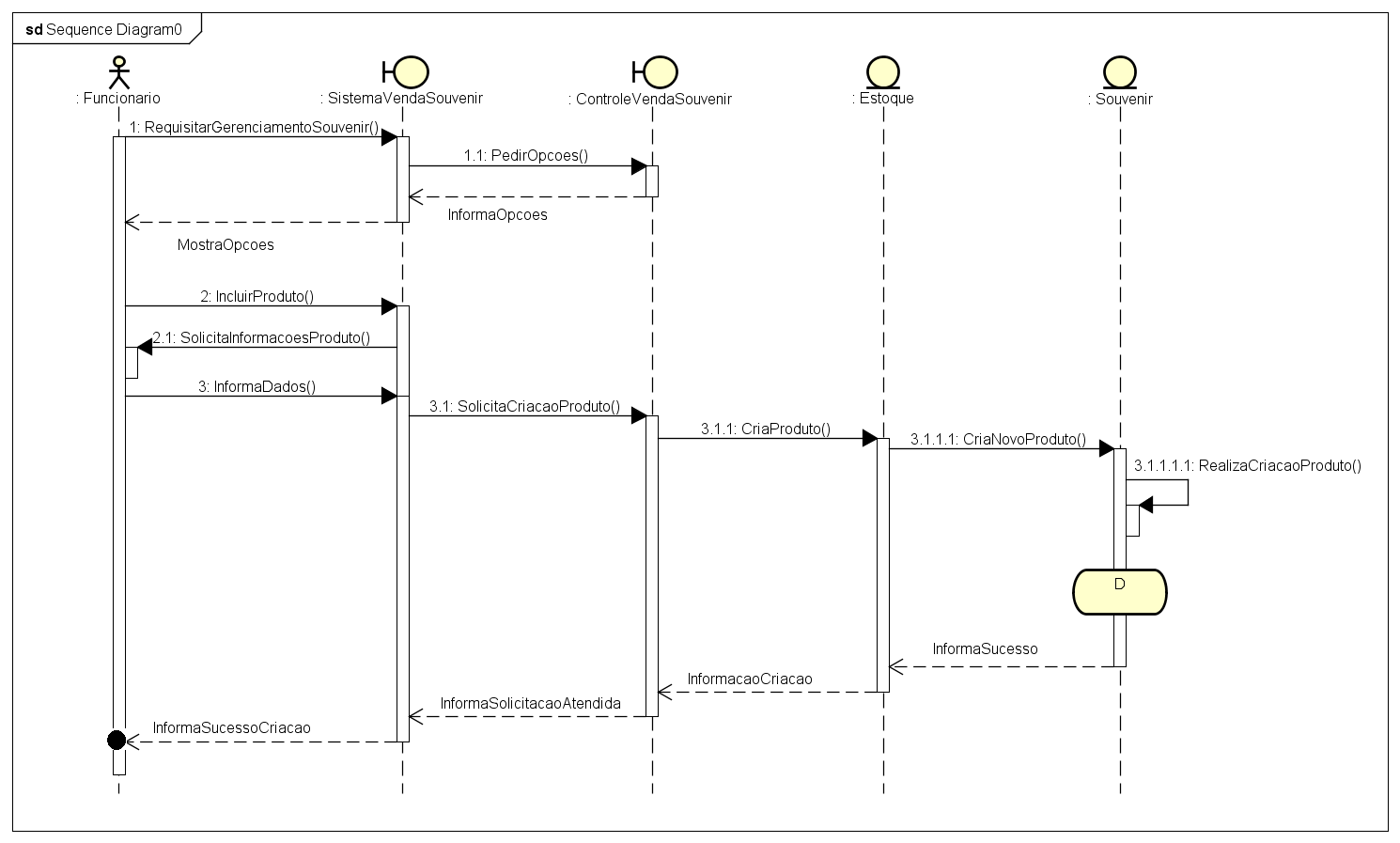
A:



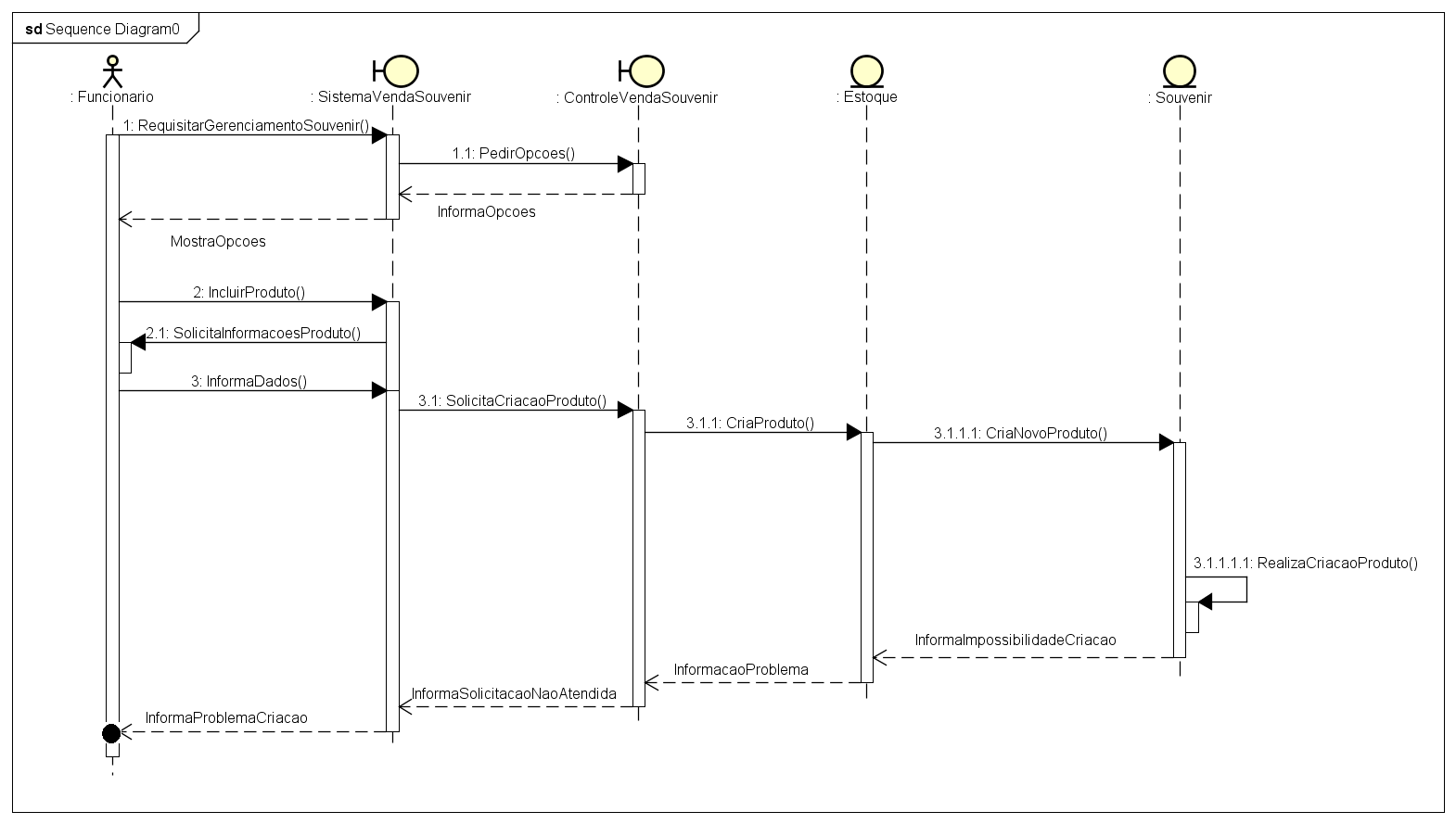
B:



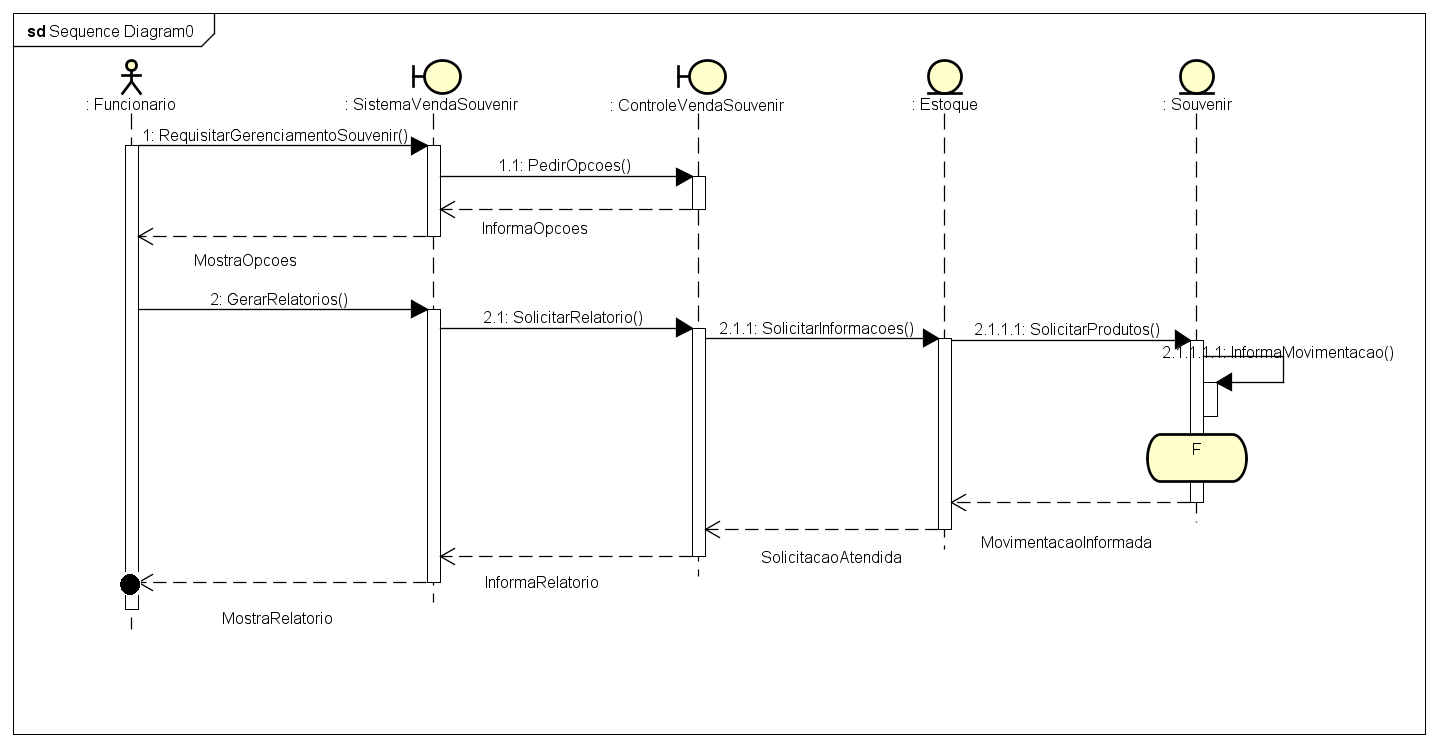
C:



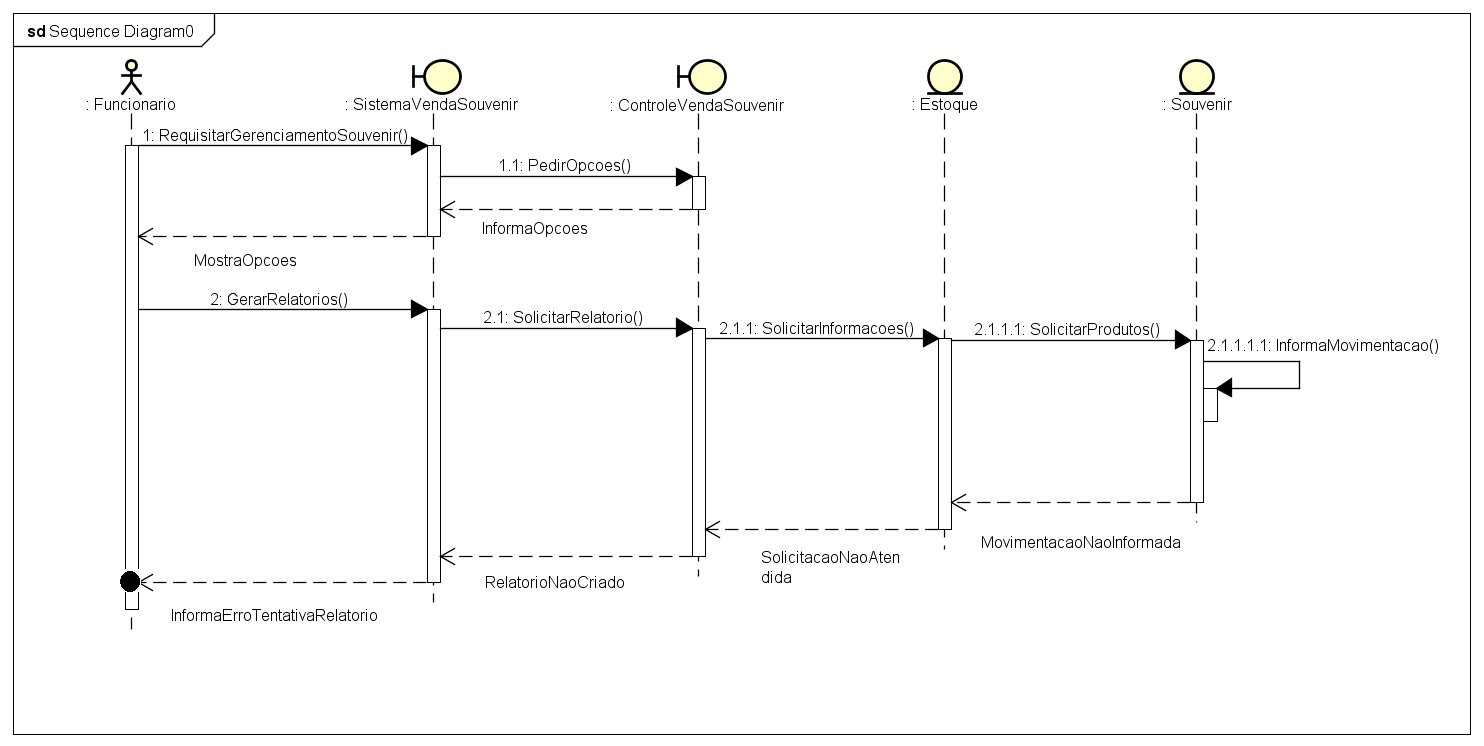
D:



E:



F:



Parte B: Atividade de Abstração

Analise a sequência de DNA abaixo. Modele um diagrama de classes utilizando delegação com relacionamento de composição que especifique o nome do filme em tempo de execução. Justifique a tua resposta.

AGGCCATTGCATTGCATGCACGATCACATGGTCACTAGGAATTCCAGTCAGTCATCCAAGGTATATCTCAGTGGCTCCTAGATTACAGACCGTACCATTAAGCCGATTACGGCTCAGTTAAGCTAATGCGCTAGTCAGTCATACTGCAAA

