

Modelagem de Transição de Estado:

9-1: Descreva a posição do diagrama de estados no processo de desenvolvimento incremental e iterativo. Quando são utilizados e para quê?

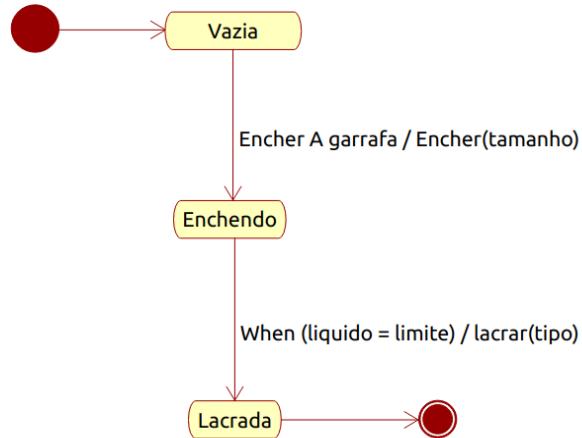
9-2: Modele por meio de um diagrama de estados a seguinte situação: em uma

máquina de encher garrafas de refrigerante passam diversas garrafas. Uma garrafa entra inicialmente vazia no equipamento. A partir de um determinado momento, ela começa a ser preenchida com refrigerante. Ela permanece nesse estado (sendo preenchida) até que, eventualmente, esteja cheia de refrigerante. Nesse momento, o equipamento sela a garrafa com uma tampinha, e assim a garrafa passa para o estado de “lacrada”. Uma garrafa vazia não deve ser lacrada pelo equipamento. Além disso, uma garrafa cheia de refrigerante não deve receber mais esse líquido.

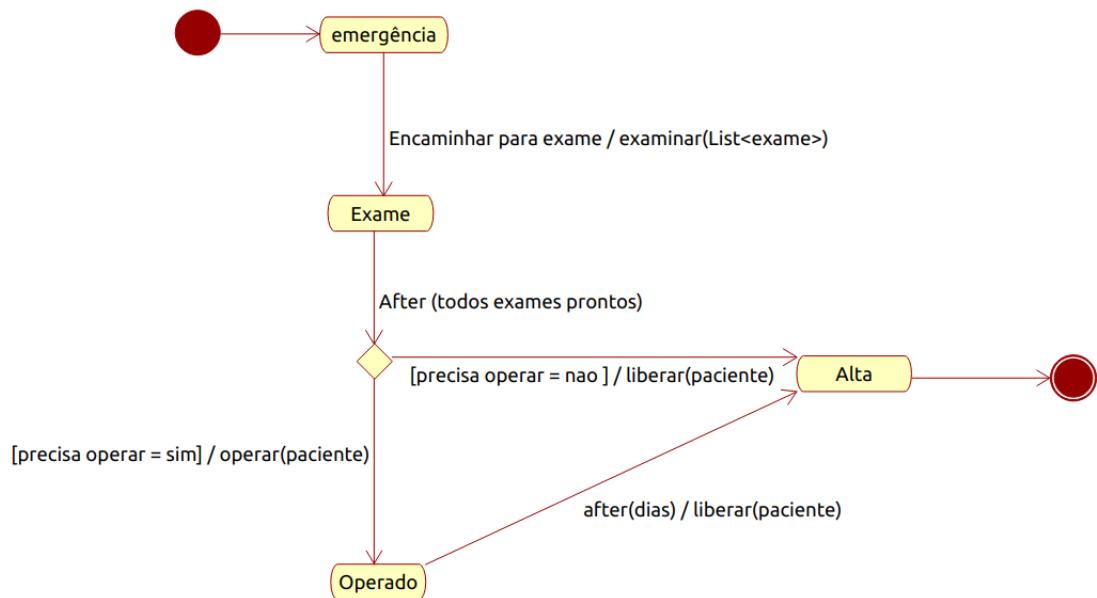
9-3: Construa um diagrama de estados considerando o seguinte “ciclo de vida” de um paciente de hospital. O paciente entra no hospital, vítima de um acidente de carro. Ele é encaminhado para a emergência. Após uma bateria de exames, esse paciente é operado. Alguns dias depois, o paciente é movido da grande emergência do hospital para a enfermaria, pois não corre mais perigo de vida. Depois de passar por um período de observação na enfermaria, o paciente recebe alta médica.

9-1 O diagrama de estados é utilizado para mostrar como os objetos de uma classe mudam de estados durante o tempo de vida dentro do software. Ele é utilizado para mostrar a mudança de estado para as classes individualmente. Durante a construção dos diagramas de transição de estados é possível refinar o diagrama de classe com novos atributos necessários para guardar o estado de um objeto, ou ainda novas operações necessárias para que a transição de estado do objeto ocorra. Além disso, esse diagrama pode contribuir com a descrição dos casos de uso, pois pode ajudar a refinar o modelo de caso de uso, adicionando etapas de transição de estados que foram descobertas no diagrama de transição de estados.

9-2



9.3



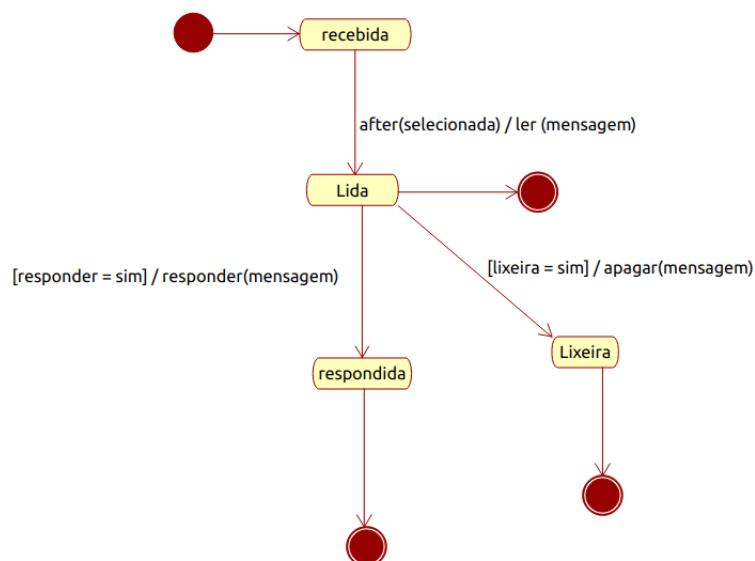
9-5: Construa um diagrama de estados para uma classe Mensagem, que representa uma mensagem de correio eletrônico. Como dica, considere os estados apresentados a seguir.

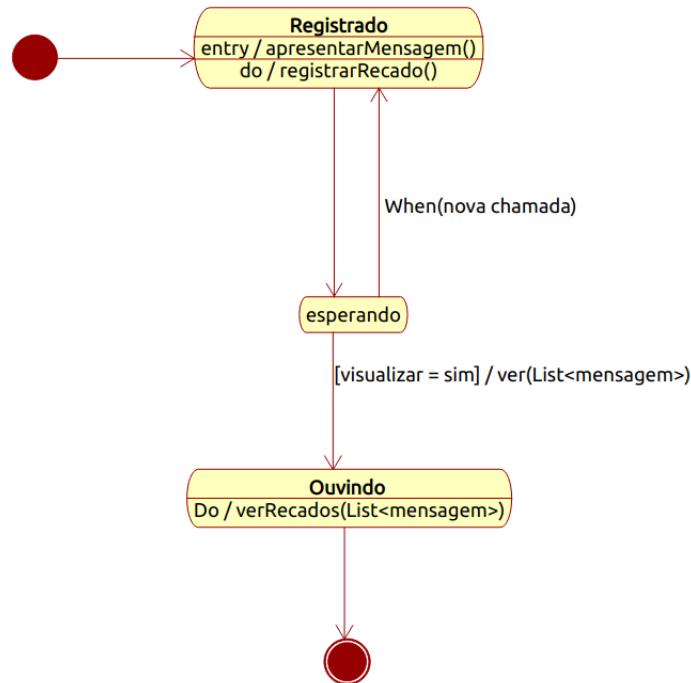
- a. Recebida: este é o estado inicial. A mensagem acabou de entrar na caixa de correio e permanece nesse estado até ser lida.
- b. Lida: a mensagem é lida pelo usuário.
- c. Respondida: o usuário responde à mensagem.
- d. Na lixeira: usuário remove a mensagem da caixa de correio.

9-6: Construa um diagrama de estados para um aparelho de secretária eletrônica. Como dica, considere os estados apresentados a seguir. Utilize estados compostos se for necessário.

- a. *Registrando recados*: alguém faz uma chamada para o aparelho de telefone ao qual a secretária está conectada, e a chamada não é atendida. A secretária, então, apresenta a mensagem gravada pelo usuário do aparelho e registra o recado deixado pela pessoa que fez a chamada telefônica.
- b. *Esperando*: a secretária está ociosa esperando ser ativada.
- c. *Revendo recados*: algum usuário da secretária requisitou que o aparelho apresentasse os recados gravados.

9.5





Arquitetura de Software:

Exercícios de Fixação [🔗](#)

1. Dada a sua complexidade, sistemas de bancos de dados são componentes relevantes na arquitetura de qualquer tipo de sistema. Verdadeiro ou falso? Justifique a sua resposta.
2. Descreva três vantagens de arquiteturas MVC.
3. Qual a diferença entre classes Controladoras em uma Arquitetura MVC tradicional e classes Controladoras de um sistema Web implementado usando um framework MVC como Ruby on Rails?
4. Descreva resumidamente quatro vantagens de microsserviços.
5. Por que microsserviços não são uma bala de prata? Isto é, descreva pelo menos três desvantagens do uso de microsserviços.
6. Explique a relação entre a Lei de Conway e microsserviços.
7. Explique o que significa desacoplamento no espaço e desacoplamento no tempo. Por que arquiteturas baseadas em filas de mensagens e arquiteturas Publish/Subscribe oferecem essas formas de desacoplamento?
8. Quando uma empresa deve considerar o uso de uma arquitetura baseada em filas de mensagens ou uma arquitetura publish/subscribe?
9. Explique o objetivo do conceito de tópicos em uma arquitetura publish/subscribe.

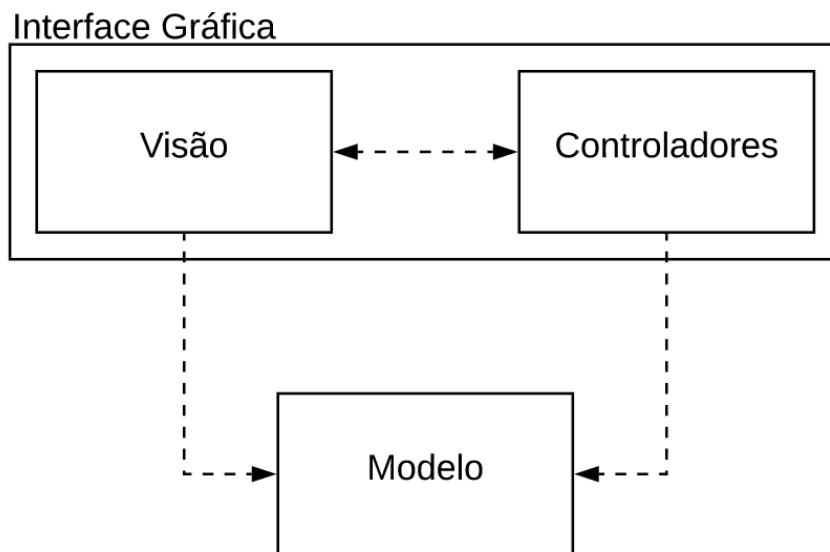
Respostas:

1: Sim, são componentes relevantes, uma vez que a escolha de um banco de dados inadequado para as necessidades do software a ser desenvolvido, poderá causar problemas futuros para o crescimento do software. Além disso, a escolha do banco precisa ser pensada, pois o suporte ao mesmo deverá ser continuado no tempo pelos seus fornecedores. Ao escolher o banco para um software deverá ser pensado questões como segurança, confiabilidade, rapidez, valor de aquisição, facilidade de uso, dentre diversas outras questões envolvidas no banco escolhido. Podemos considerar isso uma escolha arquitetural do software.

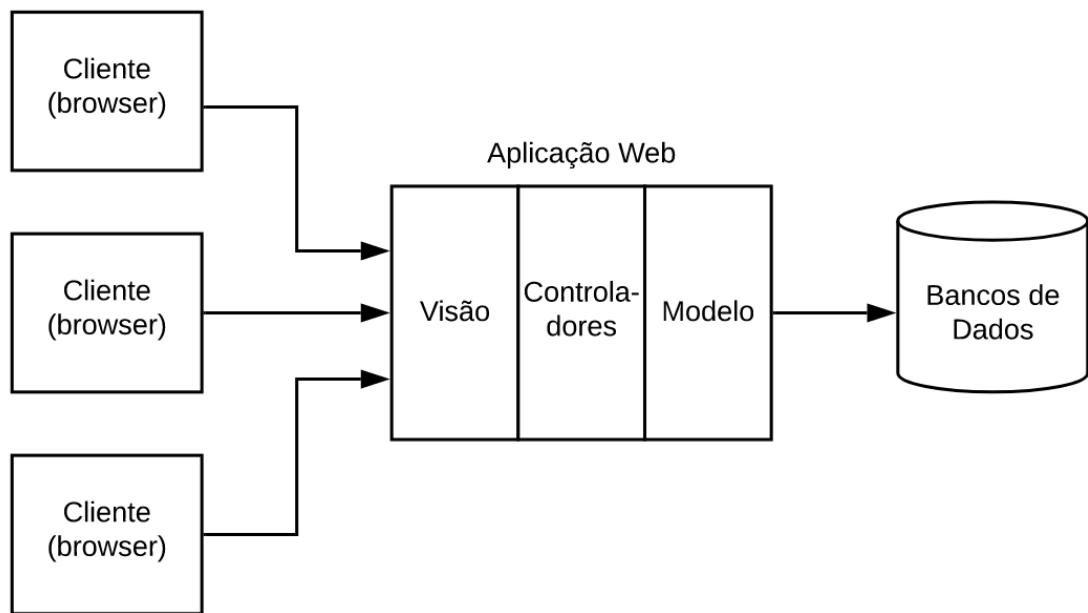
2. Seguem três justificativas:

- Desacoplamento dos componentes do software, pois ao dividir em três camadas, poderá ser realidades manutenções ou substituições em uma camada, sem prejudicar as outras camadas.
- Um padrão amplamente utilizado no mercado de software, trazendo benefícios na aquisição, por exemplo, de partes prontas do software.
- Facilidade de desenvolvimento: As equipes podem ser divididas em backend e frontend, sendo uma equipe não precisa saber do trabalho da outra, devendo existir apenas uma interface conhecida entre as parte de backend e frontend.

3. A diferença é que no MVC tradicional, por ser um sistema standalone, a controladora atuava em conjunto com a view, tornando a parte da interface gráfica, auxiliando a camada de visão a exercer suas tarefas, as duas juntas é que se comunicavam com o camada de modelo. Isso se deve ao fato de ter apenas uma visão no software, no caso o desktop onde o mesmo era instalado:



Já no MVC web, a controladora passou a não mais atuar “junto” à camada de visão, sendo que a visão passou a ter o papel de pegar os dados passados pela controladora e gerar um arquivo com tags HTML, regras CSS e scripts Javascript, que depois são enviados pela visão para o browser do usuário que solicitou tal página. Isso faz com que possa existir várias camadas de visão diferente, cada uma respondendo de maneira diferente para cada cliente que solicitou os serviços do software:



4. vantagens dos micro-serviços:

- (1) eles permitem a evolução mais rápida e independente de um sistema, permitindo que cada time tenha seu próprio regime de liberação de novas releases
- (2) eles permitem escalar um sistema em um nível de granularidade mais fino do que é possível com monólitos
- (3) Como os microsserviços são autônomos e independentes eles podem ser implementados em tecnologias diferentes
- (4) arquiteturas baseadas em microsserviços podem ter **falhas parciais**.

5: desvantagens dos micro-serviços

- (1) Complexidade: quando dois módulos executam em um mesmo processo, a comunicação entre eles é por meio de chamadas de métodos. Quando esses módulos estão em máquinas diferentes, a comunicação entre eles deve usar algum protocolo de comunicação, como **HTTP/REST**. Ou seja, os desenvolvedores terão que dominar e usar um conjunto de tecnologias para comunicação em redes.
- (2) Latência: a comunicação entre microsserviços também envolve um atraso maior, que chamamos de **latência**. Quando um cliente chama um método em um sistema monolítico, a latência é mínima. Por exemplo, é raro um desenvolvedor deixar de

usar uma chamada de método apenas para melhorar o desempenho de seu sistema. Porém, esse cenário muda quando o serviço chamado está em uma outra máquina, talvez do outro lado do planeta no caso de uma empresa global. Nessas situações, existe um custo de comunicação que não é desprezível. Qualquer que seja o protocolo de comunicação usado, essa chamada terá que passar pelo cabo da rede — ou pelo ar e pela fibra ótica — até chegar à máquina de destino.

- (3) Transações Distribuídas: Como vimos, microsserviços devem ser autônomos também do ponto de vista de dados. Isso torna mais complexo garantir que operações que operam em dois ou mais bancos de dados sejam atômicas, isto é, ou elas executam com sucesso em todos os bancos ou então falham.

6: Microsserviços constituem um exemplo de aplicação da **Lei de Conway**. Formulada em 1968 por Melvin Conway, ela é uma das leis empíricas sobre desenvolvimento de software, assim como a Lei de Brooks, que estudamos no Capítulo 1. A Lei de Conway afirma o seguinte: empresas tendem a adotar arquiteturas de software que são cópias de suas estruturas organizacionais. Em outras palavras, a arquitetura dos sistemas de uma empresa tende a espelhar seu organograma. Por isso, não é coincidência que microsserviços sejam usados, principalmente, por grandes empresas de Internet que possuem centenas de times de desenvolvimento distribuídos em diversos países. Além de descentralizados, esses times são autônomos e sempre incentivados a produzir inovações.

7:

- **Desacoplamento no espaço:** clientes não precisam conhecer os servidores e vice-versa. Em outras palavras, o cliente é exclusivamente um produtor de informações. Mas ele não precisa saber quem vai consumir essa informação. O raciocínio inverso vale para os servidores.
- **Desacoplamento no tempo:** clientes e servidores não precisam estar simultaneamente disponíveis para se comunicarem. Se o servidor estiver fora do ar, os clientes podem continuar produzindo mensagens e colocando-as na fila. Quando o servidor voltar a funcionar, ele irá processar essas mensagens.

Desacoplamento no espaço torna soluções baseadas em filas de mensagens bastante flexíveis. Os times de desenvolvimento — tanto do sistema cliente, como do sistema servidor — podem trabalhar e evoluir seus sistemas com autonomia. Atrasos de um time não travam a evolução de outros times, por exemplo. Para isso, basta que o formato das mensagens permaneça estável ao longo do tempo. Já desacoplamento no tempo torna a solução robusta a falhas. Por exemplo, quedas do servidor não têm impacto nos clientes. No entanto, é importante que o broker de mensagens seja estável e capaz de armazenar uma grande quantidade de mensagens. Para garantir a disponibilidade desses brokers, eles costumam ser gerenciados pelos times de infraestrutura básica das empresas.

Filas de mensagens permitem também escalar mais facilmente um sistema distribuído. Para isso, basta configurar múltiplos servidores consumindo mensagens da mesma fila

8:

- Uma empresa deve considerar uma arquitetura baseada em filas, quando o número de requisições a seus servidores é muito grande em um mesmo espaço de tempo, sendo que a empresa pode alocar maior número de servidores para atender as requisições de forma dinâmica. Além disso, existe a questão do serviço nunca estar forma, uma vez que o primeiro a atender a requisição é o broker, e posteriormente o servidor. Caso um servidor quebre, poderá ser substituído rapidamente por outro, sem o cliente perceber.
- No caso da arquitetura Publish/Subscribe, a empresa deve considerar que as funcionalidades do software passaram a ter uma ideia de um serviço que é oferecido a seus clientes, sem necessariamente um conhecer o outro, pois existe um local comum, onde o serviço está disponível e o cliente poderá se inscrever para usá-lo. Basta que a comunicação entre ambos seja padronizada, como por exemplo, usando XML. Também deve considerar que um serviço, quando houver mudanças, irá notificar a todos os que o assinaram, sem a necessidade do cliente ter que consultar o serviço para ver as mudanças.

9: Em alguns sistemas publish/subscribe, eventos são organizados em **tópicos**, que funcionam como categorias de eventos. Quando um publicador produz um evento, ele deve informar seu tópico. Assim, clientes não precisam assinar todos os eventos que ocorrem no sistema, mas apenas eventos de um determinado tópico.

Arquitetura de Software:

11.1: Explique o que é arquitetura de software, o que deseja-se estabelecer ao apontar a arquitetura do mesmo. Explique o que são decisões arquitetônicas em um software e como isso impacta o mesmo.

Arquitetura de software é algo difícil de estabelecer por completo, porém podemos estabelecer a arquitetura de um SW como as decisões sobre os requisitos não funcionais, a forma como o mesmo é dividido em componentes, camadas e nós físicos. Porém o assunto não se limita apenas a isso, uma vez que arquitetar um software é algo que envolve a experiência do arquiteto, o tamanho do software e seu domínio de negócios.

Decisões arquitetônicas e tudo aquilo que envolve basicamente as decisões acerca dos requisitos não funcionais, e formas de divisão física e lógica do mesmo. Essas decisões são de grande importância, pois decisões erradas poderão impactar no futuro do sistema de maneira negativa, e talvez, não possam ser contornadas posteriormente.

11.2 Explique a diferença entre arquitetura lógica e física de um software. Qual o objetivo de cada uma dessas arquitetura e como demonstrá-las.

A arquitetura lógica é uma forma de dividir o sistema em componentes maiores, que normalmente seguem a lei de Conway, onde os componentes de um software tendem a refletir a divisão setorial das organizações. Podemos usar o diagrama de componentes da uml para demonstrar essa arquitetura.

A arquitetura física é a forma como os componentes do software são divididos fisicamente. Nessa divisão deve ser detalhado qual o hardware envolvido na estrutura do software para o mesmo funcionar. Chamados esses componentes de hardware de nós da arquitetura física. Para demonstrar essa arquitetura podemos utilizar o diagrama de implantação da uml.

11.3. Explique o que é uma arquitetura fechada e uma arquitetura aberta. O que isso tem a ver com camadas de software ?

Um software pode ter seus componentes divididos em camadas, que representam um objetivo comum dos componentes contidos na mesma. Nesse sentido de divisão de componentes em camadas, podemos classificar a dependência entre as mesmas. Quando dizemos que uma arquitetura é fechada, significa que as camadas do software possuem uma hierarquia, sendo que as camadas uma camada mais alta depende de outra subjacente. Camadas mais baixas têm a características de não serem dependentes das camadas mais altas.

Quando falamos de uma arquitetura aberta, é quando não existe dependência entre as camadas, qualquer camada pode acessar ou depender de qualquer outra.

Diagrama de Componentes :

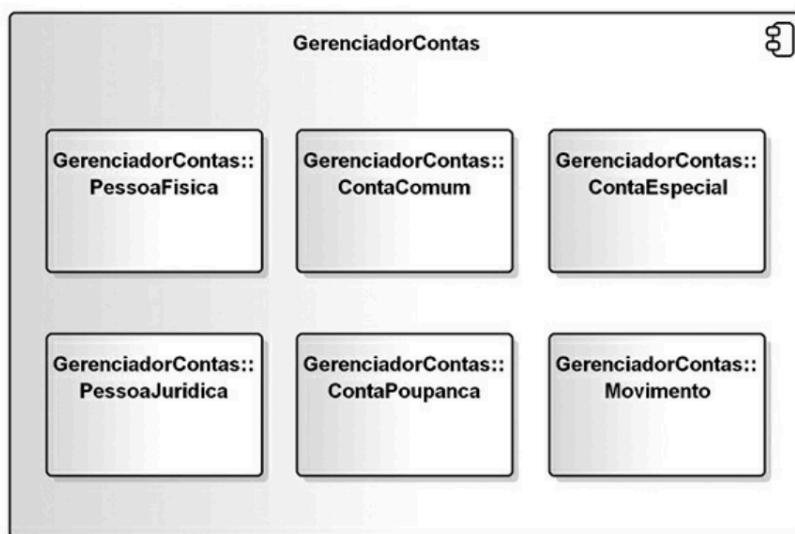
12.1: Explique qual o principal objetivo do diagrama de componentes.

O objetivo deste diagrama é mostrar como é a arquitetura lógica de um software. Mostra como os componentes se conectam uns com os outros.

12.2 Qual a diferença entre componentes de caixa branca e componentes de caixa preta ? Apresente um exemplo para cada tipo (caixa branca e caixa preta).

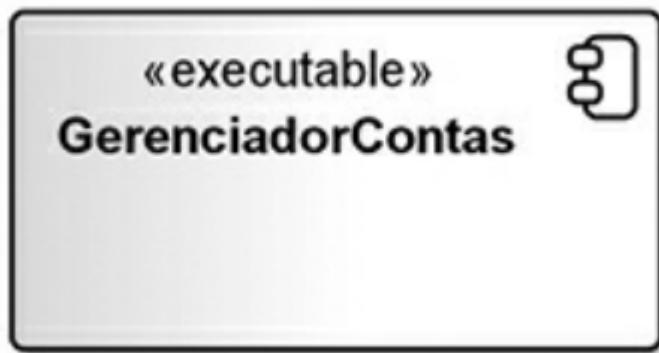
Os componentes de caixa branca são apresentados com as classes que o compõem sendo apresentadas em seu interior.

ex:



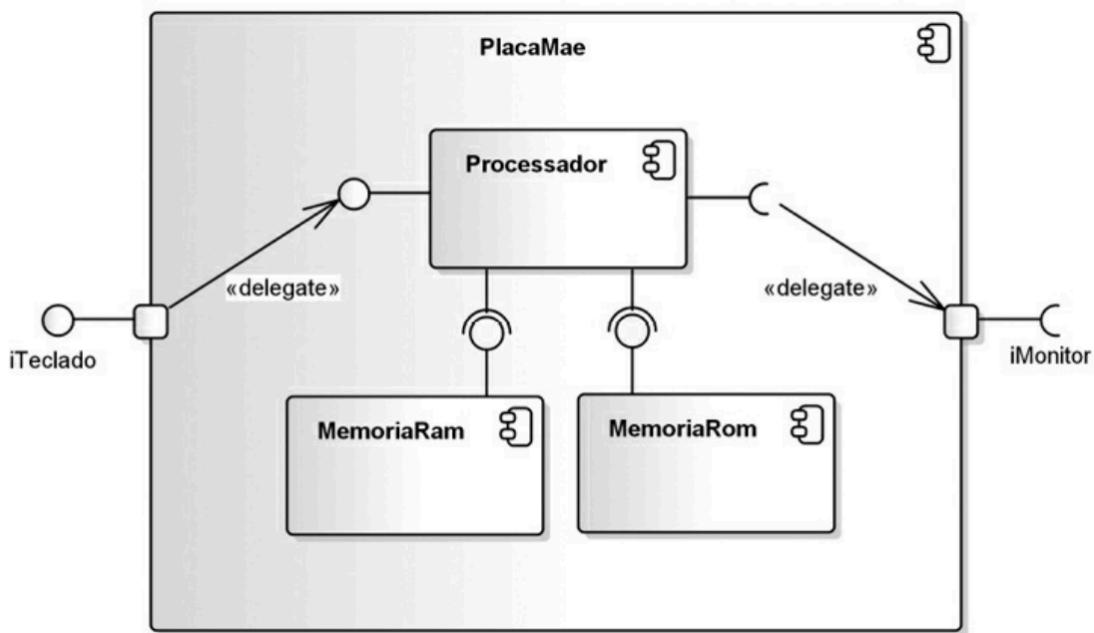
Os componentes de caixa preta é quando as classes que fazem parte do mesmo são omitidas da sua representação. Sendo apresentado apenas o componente em si.

ex:



12.3 Explique qual o objetivo das portas no diagrama de componentes.
Apresenta uma modelagem de exemplo.

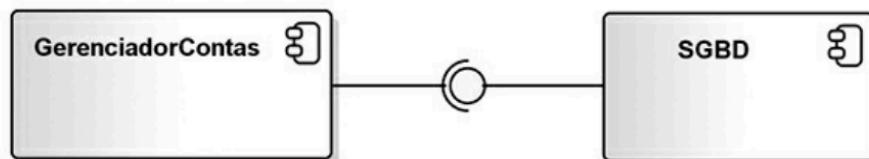
As portas são usadas para demonstrar a comunicação de um componente interno de um sistema com o ambiente externo ao software. Podemos representar as portas com interfaces que são fornecidas pelo componente ao mundo externo, para que componentes externos possam usar os componentes internos do software. Também é possível que as portas tenham interfaces requeridas, neste caso, é a forma como o componente interno se comunica com partes externas do software para resolver algum problema interno do componente.



12.4 Explique o que são interfaces requeridas e interfaces fornecidas no diagrama de componentes. Como eles são modelados no diagrama de componentes.

É a forma de demonstrar a comunicação entre os componentes de um software, e demonstra a dependência existente entre os mesmos. Quando um componente tem uma interface fornecida significa que o mesmo oferece seus serviços para componentes que desejam usá-los. Quando um componente tem um interface requerida significa que o mesmo se acopla a um outro componente que oferece um serviço que o mesmo necessita.

Ex:

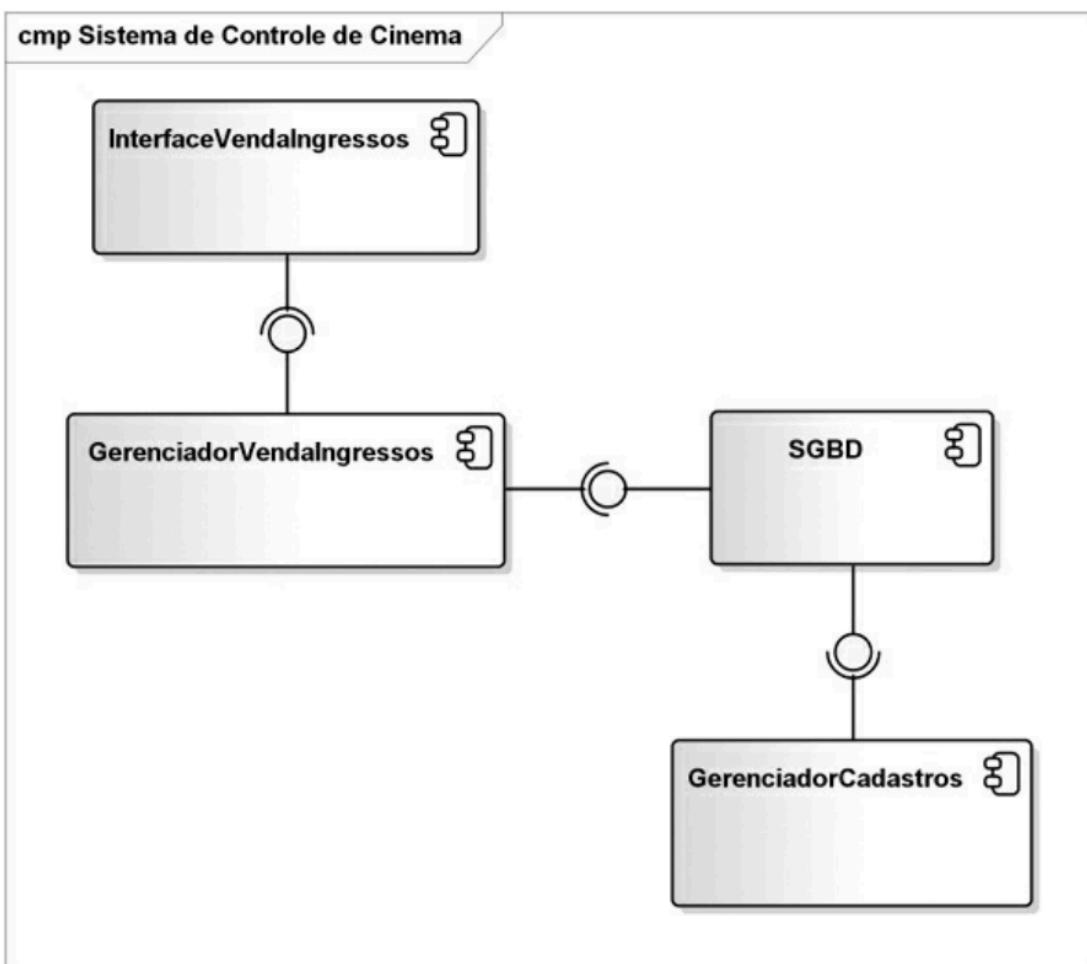


Neste caso, o SGBD tem uma interface fornecida, e que é utilizada pelo componente GerenciadorContas, que tem uma interface requerida que se acopla ao SGBD.

12.5.1 Sistema de Controle de Cinema

Desenvolva o diagrama de componentes para um sistema de controle de cinema, sabendo que:

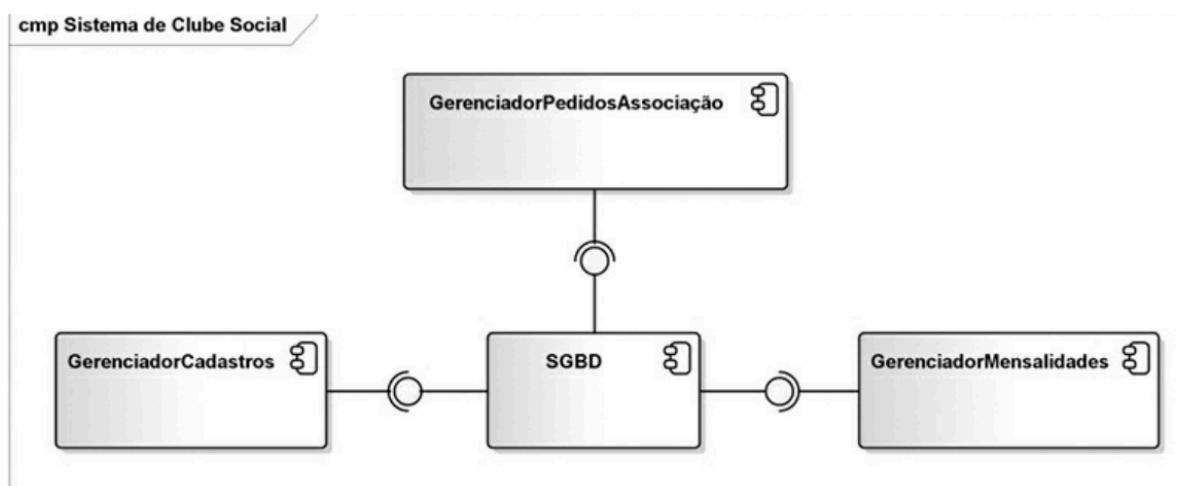
- É preciso existir um módulo para gerir a venda de ingresso aos clientes. Esse módulo deve gerar os ingressos e emiti-los por meio da interface (que pode ser física também) para os clientes do cinema.
- O sistema necessita de um SGBD para persistir suas informações.
- Finalmente, é necessário um módulo de manutenção do sistema, onde basicamente serão mantidos os cadastros de sessões, salas, filmes,



12.5.2 Sistema de Controle de Clube Social

Desenvolva o diagrama de componentes referente ao sistema de clube social, levando em consideração os seguintes fatos:

- O clube necessita de um módulo para gerenciar os pedidos de associação dos candidatos a sócio.
- O clube precisa também de um módulo para manter o cadastro de seus sócios, dependentes e suas categorias.
- É preciso existir um SGBD para persistir esses dados.
- Finalmente, é necessário haver um módulo para a emissão e quitação das mensalidades dos sócios.

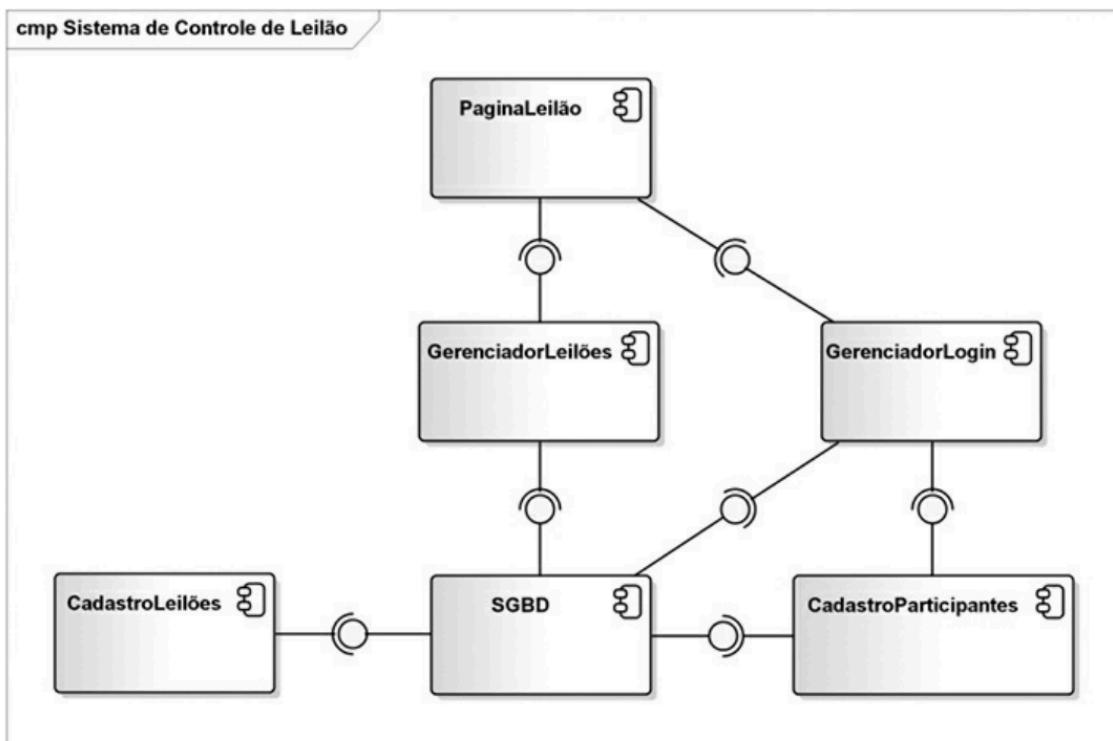


12.5.4 Sistema para Controle de Leilão Via Internet

Desenvolva o diagrama de componentes relativo a um sistema de controle de leilão via internet, de acordo com os seguintes requisitos:

- O sistema necessita de uma página por meio da qual os participantes poderão se logar ou se autorregular. Essa página será útil também para apresentar os itens anunciados pelo leiloeiro e receber as ofertas dos participantes, bem como para anunciar um item como arrematado.
- O sistema precisa de um módulo que gerencie o login dos participantes. É recomendável existir um módulo associado ao módulo de login para permitir o autorregistro das pessoas interessadas em participar do leilão.

- É necessário haver um módulo que permita registrar novos leilões, bem como cadastrar os itens que serão anunciados neles.
- Também é necessário um módulo responsável por gerenciar cada leilão, que permita a um leiloeiro abrir um leilão, anunciar os itens deste, receber lances, anunciar vencedores e arrematar itens.
- Finalmente, é necessário um sistema gerenciador de banco de dados para persistir e recuperar as informações necessárias ao sistema.



12.5.6 Sistema de Controle de Imobiliária

Desenvolva o diagrama de componentes para um sistema de controle de imobiliária, de acordo com as seguintes afirmações:

- Como nos outros exercícios, é preciso haver um módulo de manutenção dos cadastros do sistema.
- O software também necessita de um módulo que permita aos usuários se autenticarem.
- O sistema deve conter ainda um módulo para gerenciar os contratos que autorizam a imobiliária a vender ou alugar imóveis.
- Finalmente, o sistema deve conter um módulo para gerenciar as vendas de imóveis realizadas pela imobiliária, bem como um módulo que gerencie as locações de um imóvel por inquilinos e os pagamentos

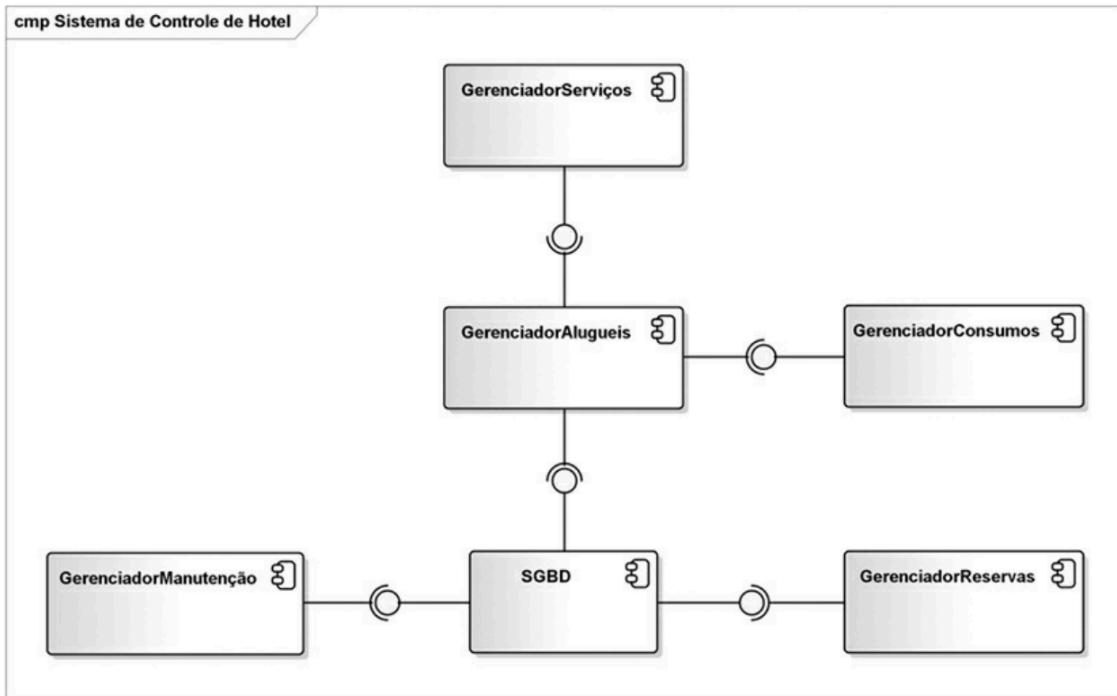


Diagrama de Implantação

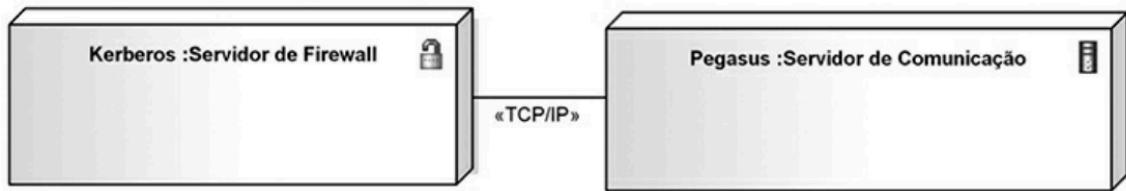
13.1 O que são os nós no diagrama de implantação. Apresente os tipos de nós possíveis e seus respectivos estereótipos.

Os nós em um diagrama de implantação são os itens de hardware envolvidos na arquitetura física do software. Esses nós devem se comunicar, cada um com sua função, para oferecer os serviços do software adequadamente. Os estereótipos possíveis são: <<device>> que é um nó genérico. O <<computer>> que representa um computador simples, como por exemplo a máquina do cliente. <<secure>> que representa um item de hardware de segurança. <<server>> que representa um servidor e o <<store>> que representa onde os dados ficam guardados, exemplo, a máquina onde fica o SGBD.

13.2 Como são as associações entre os nós no diagrama de componentes e como podemos especificar os meios de comunicação entre os mesmos ?

São representadas por uma linha contínua que liga um nó ao outro. Essa ligação demonstra como um nó se comunica com outro fisicamente. A especificação se dá através de um estereótipo que aponta qual o protocolo de comunicação utilizado na ligação física.

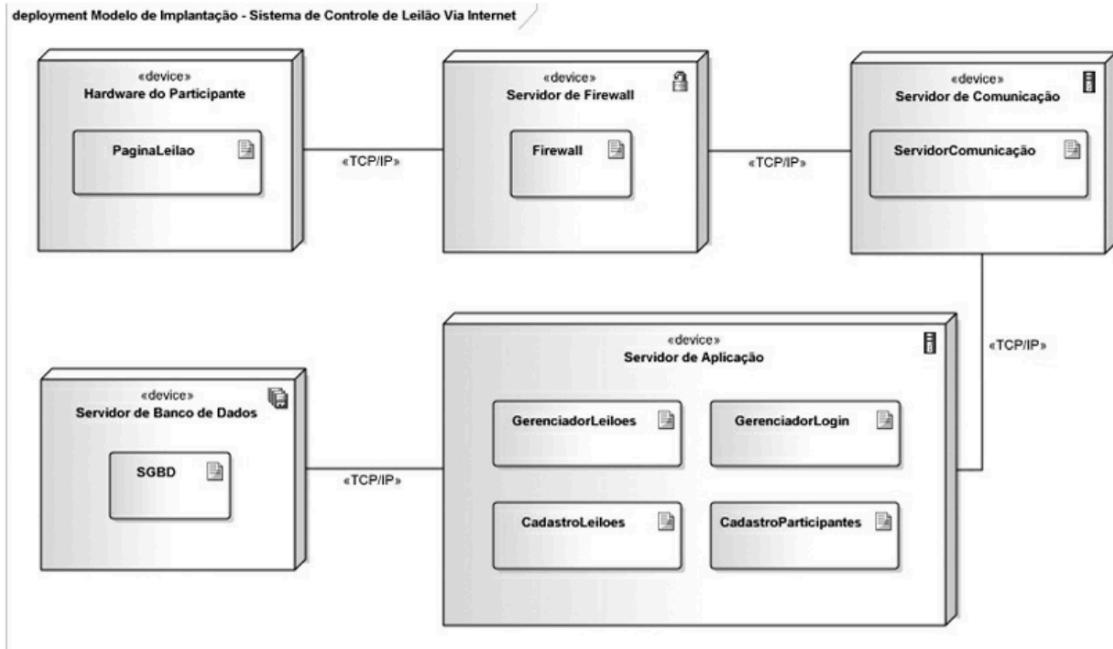
ex:



13.9.1 Sistema para Controle de Leilão Via Internet

Desenvolva o diagrama de implantação para o sistema de controle de leilão via internet, sabendo que:

- Os participantes acessam o sistema de leilão de suas próprias máquinas pessoais, de forma que é necessário representá-las.
- Uma vez que esse sistema é acessado externamente, é necessário estabelecer uma linha de segurança para impedir invasões.
- O sistema precisa suportar uma quantidade potencialmente grande de conexões. Assim, é necessário existir um servidor de comunicação.
- Toda a aplicação pode rodar em um único servidor, não havendo necessidade de dividi-la.
- No entanto, deve haver um servidor de banco de dados para armazenar as informações do sistema, bem como as recuperar quando solicitado. Embora este pudesse estar instalado no mesmo hardware, poderia ser útil se fosse executado em outra máquina.



Capítulo 5 - Propriedades e princípios de projeto de software:

1. Descreva três benefícios da propriedade de projeto chamada ocultamento de informação (*information hiding*)?
2. Suponha que um programador adote a seguinte estratégia: ao implementar qualquer nova funcionalidade ou corrigir um bug que implique na modificação de duas classes A e B localizadas em arquivos diferentes, ele conclui a tarefa movendo as classes para o mesmo arquivo. Explicando melhor: após terminar a tarefa de programação que ficou sob sua responsabilidade, ele escolhe uma das classes, digamos a classe B, e a move para o mesmo arquivo da classe A. Agindo dessa maneira, ele estará melhorando qual propriedade de projeto? Por outro lado, qual propriedade de projeto estará sendo afetada de modo negativo? Justifique.

Resposta:

1-) Seguem os benefícios:

- Facilidade de implementação: pois possibilita o desenvolvimento em paralelo de componentes, pois a única coisa que conhecida externamente aos componentes é sua interface, que deve ser projetada para ser intuitiva e mantida durante o tempo
- Flexibilidade a mudanças: pois como os módulos ocultam sua complexibilidade, mudanças feitas no mesmo não afetam outros componentes, desde que não mudem suas interfaces conhecidas.
- Facilidade de entendimento: Como os componentes ocultam sua complexibilidade, oferecendo apenas uma interface intuitiva, fica mais fácil o entendimento do mesmo, pois devemos apenas saber usar sua interface.

2-) Ele estará eliminando o problema relacionado ao acoplamento entre as classes, uma vez que não existem mais ligações a serem feitas, tudo estará no mesmo lugar. Entretanto poderá estar cometendo um grande erro de coesão, pois estará juntando operações e métodos que podem não ter relação, deixando a nova classe sem coesão, objetividade e difícil de entender seus serviços.

4. Defina: (a) acoplamento aceitável; (b) acoplamento ruim; (c) acoplamento estrutural; (d) acoplamento evolutivo (ou lógico).
5. Dê um exemplo de: (1) acoplamento estrutural e aceitável; (2) acoplamento estrutural e ruim.
6. É possível que uma classe A esteja acoplada a uma classe B sem ter uma referência para B em seu código? Se sim, esse acoplamento será aceitável ou será um acoplamento ruim?
7. Suponha um programa em que todo o código está implementado no método `main`. Ele tem um problema de coesão ou acoplamento? Justifique.

Respostas :

4-)

Dizemos que existe um **acoplamento aceitável** de uma classe A para uma classe B quando:

- A classe A usa apenas métodos públicos da classe B.
- A interface provida por B é estável do ponto de vista sintático e semântico. Isto é, as assinaturas dos métodos públicos de B não mudam com frequência; e o mesmo acontece como o comportamento externo de tais métodos. Por isso, são raras as mudanças em B que terão impacto na classe A.

Por outro lado, existe um **acoplamento ruim** de uma classe A para uma classe B quando mudanças em B podem facilmente impactar A. Isso ocorre principalmente nas seguintes situações:

- Quando a classe A realiza um acesso direto a um arquivo ou banco de dados da classe B.
 - Quando as classes A e B compartilham uma variável ou estrutura de dados global. Por exemplo, a classe B altera o valor de uma variável global que a classe A usa no seu código.
 - Quando a interface da classe B não é estável. Por exemplo, os métodos públicos de B são renomeados com frequência.
-
- **Acoplamento estrutural** entre A e B ocorre quando uma classe A possui uma referência explícita em seu código para uma classe B. Por exemplo, o acoplamento entre `Estacionamento` e `Hashtable` é estrutural.

- **Acoplamento evolutivo (ou lógico)** entre A e B ocorre quando mudanças na classe B tendem a se propagar para a classe A. No exemplo mencionado, no qual a classe A depende de um inteiro armazenado em um arquivo interno de B, não existe acoplamento estrutural entre A e B, pois A não declara nenhuma variável do tipo B, mas existe acoplamento evolutivo. Por exemplo, mudanças no formato do arquivo criado por B terão impacto em A.

5-)

- Acoplamento estrutural aceitável:

<pre>class B { int total; public int getTotal() { return total; } private void g() { // computa valor de total File f = File.open("arq1"); f.writeInt(total); ... } }</pre>		<pre>class A { private void f(B b) { int total; total = b.getTotal(); ... } }</pre>	
---	--	---	--

- Acoplamento estrutural ruim:

```

import java.util.Hashtable;

public class Estacionamento {

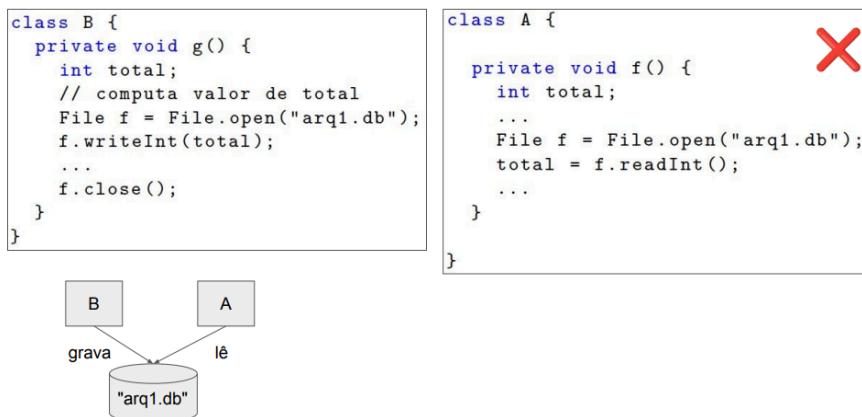
    public Hashtable<String, String> veiculos;

    public Estacionamento() {
        veiculos = new Hashtable<String, String>();
    }

    public static void main(String[] args) {
        Estacionamento e = new Estacionamento();
        e.veiculos.put("TCP-7030", "Uno");
        e.veiculos.put("BNF-4501", "Gol");
        e.veiculos.put("JKL-3481", "Corsa");
    }
}

```

6-) Sim é possível por uma referência indireta. Podemos classificar como uma dependência lógica, e não estrutural. Ex:



7-) Um problema de coesão, pois não usa as propriedades da orientação a objetos. Não existe uma divisão de responsabilidades entre classes. A abstração está com um erro grave ao colocar todo o software em apenas uma classe main.

8. Qual princípio de projeto é violado pelo seguinte código?

```
void onclick() {
    num1 = textfield1.value();
    c1 = BD.getConta(num1)
    num2 = textfield2.value();
    c2 = BD.getConta(num2)
    valor = textfield3.value();
    beginTransaction();
    try {
        c1.retira(valor);
        c2.deposita(valor);
        commit();
    }
    catch() {
        rollback();
    }
}
```

Fere o “**princípio da responsabilidade única**” uma vez que a classe acima trata dos eventos da view, e também do acesso ao banco de dados. Deveria haver uma separação entre essas funções em classes distintas.

9. Costuma-se afirmar que existem três conceitos chaves em orientação a objetos: encapsulamento, polimorfismo e herança. Suponha que você tenha sido encarregado de projetar uma nova

linguagem de programação. Suponha ainda que você poderá escolher apenas dois dos três conceitos que mencionamos. Qual dos conceitos eliminaria então da sua nova linguagem? Justifique sua resposta.

Seria possível retirar a herança, uma vez que existe um princípio de projeto chamado “Prefira Composição a Herança”. A herança possui o problema de expor a classe pai para as filhas. Além disso, existem situações em que o desperdício pode existir, quando as filhas não usam todos os serviços da classe pai, ou ainda, quando a herança existe apenas para representar papéis. Portanto, manteria a propriedade de encapsulamento e polimorfismo, pois são fundamentais na O.O.

10. Qual princípio de projeto é violado pelo seguinte código? Como você poderia alterar o código do método para atender a esse princípio?

```
void sendMail(ContaBancaria conta, String msg) {
    Cliente cliente = conta.getCliente();
    String endereco = cliente.getMailAddress();
    "Envia mail"
}
```

Viola o princípio de deméter, uma vez que acessa o cliente através da classe ContaBancaria, ou seja, não tem um acesso direto ao método getmailaddress().

Viola o princípio da intervenção de dependência, uma vez que o cliente acessa a classe ContaBancaria diretamente, e não uma interface.

11. Qual princípio de projeto é violado pelo seguinte código? Como você poderia alterar o código do método para atender a esse princípio?

```
void imprimeDataContratacao(Funcionario func) {  
    Date data = func.getDataContratacao();  
    String msg = data.format();  
    System.out.println(msg);  
}
```

Viola o princípio de deméter, uma vez que está fazendo mais de uma chamada para fazer determinada situação, que é formatar uma data para string. Uma alternativa seria já passar para a classe a data no formato de string para poder ser utilizada.

12. As pré-condições de um método são expressões booleanas envolvendo seus parâmetros (e, possivelmente, o estado de sua classe) que devem ser verdadeiras antes da sua execução. De forma semelhante, as pós-condições são expressões booleanas envolvendo o resultado do método.

Considerando essas definições, qual princípio de projeto é violado pelo código abaixo?

```
class A {  
    int f(int x) { // pre: x > 0  
        ...  
        return exp;  
    } // pos: exp > 0  
    ...  
}
```

```
class B extends A {  
    int f(int x) { // pre: x > 10  
        ...  
        return exp;  
    } // pos: exp > -50  
    ...  
}
```

Viola o princípio de substituição de Liskov uma vez que a classe B possui aceita apenas valores de pré-condições menores que a classe A. Enquanto A aceita valores maiores a 0, a classe que só aceita valores maiores a 10. A aceitação de classe B é mais restritiva que a classe A.

13: Explique as propriedades de integridade conceitual, e porque ela é considerada como essencial em um software. Apresente um exemplo de modelagem de classe, e um contra-exemplo.

Integridade conceitual prega que os componentes de um sistema devem ter uma padronização para que seus usuários entendam um padrão de utilização para o mesmo.

Integridade Conceitual = coerência e padronização de funcionalidades, projeto e implementação



Exemplo



Contra-Exemplo

10

14: Explique as vantagens e desvantagens de modularizar um sistema, qual a relação existente entre as propriedades de coesão e acoplamento, e da propriedade de ocultação de informação. Apresente um exemplo de modelagem de classe, e um contra-exemplo.

Modularizar um sistema possui a vantagem de facilitar o entendimento do mesmo, além de facilitar sua implementação por equipes diferentes. Fazer isso aumenta a coesão do sistema. As desvantagens são o aumento do acoplamento, pois aumenta a necessidade dos módulos terem que se comunicar em uma relação de dependência.

Exemplo de módulo coeso e adequado:

Exemplo

```
class Stack<T> {  
    boolean empty() { ... }  
    T pop() { ... }  
    push (T) { ... }  
    int size() { ... }  
}
```



Contra-exemplo:

Contra-exemplo 1

```
float sin_or_cos(double x, int op) {  
    if (op == 1)  
        "calcula e retorna seno de x"  
    else  
        "calcula e retorna cosseno de x"  
}
```



15: Explique os conceitos a seguir. Para cada item, apresente um exemplo prático, e um contra-exemplo.

- A. Responsabilidade Única
- B. Segregação de Interfaces
- C. Inversão de Dependências
- D. Prefira Composição a Herança
- E. Demeter
- F. Aberto/Fechado
- G. Substituição de Liskov

A-

5.6.1 Princípio da Responsabilidade Única

Esse princípio é uma aplicação direta da ideia de coesão. Ele propõe o seguinte: toda classe deve ter uma única responsabilidade. Mais ainda, responsabilidade, no contexto do princípio, significa "motivo para modificar uma classe". Ou seja, deve existir um único motivo para modificar qualquer classe em um sistema.

exemplo:

```

class Console {
    
    void imprimeIndiceDesistencia(Disciplina disciplina) {
        double indice = disciplina.calculaIndiceDesistencia();
        System.out.println(indice);
    }
}

class Disciplina {
    double calculaIndiceDesistencia() {
        double indice = "calcula índice de desistência"
        return indice;
    }
}

```

Contra-exemplo:

```

class Disciplina {
    
    void calculaIndiceDesistencia() {
        indice = "calcula índice de desistência"
        System.out.println(indice);
    }
}

```

B -

5.6.2 Princípio da Segregação de Interfaces

Assim como o princípio anterior, esse princípio é uma aplicação da ideia de coesão. Melhor dizendo, ele é um caso particular de Responsabilidade Única com foco em interfaces. O princípio define que interfaces têm que ser pequenas, coesas e, mais importante ainda, específicas para cada tipo de cliente. O objetivo é evitar que clientes dependam de interfaces com métodos que eles não vão usar. Para evitar isso, duas ou mais interfaces específicas podem, por exemplo, substituir uma interface de propósito geral.

Exemplo:

```

interface Funcionario {
    double getSalario();
    ...
}

interface FuncionarioCLT extends Funcionario {
    double getFGTS();
    ...
}

interface FuncionarioPublico extends Funcionario {
    int getSIAPE();
    ...
}

```

contra-exemplo:

```

interface Funcionario {
    double getSalario();

    double getFGTS(); // apenas funcionários CLT

    int getSIAPE(); // apenas funcionários públicos
    ...
}

```

C -

5.6.3 Princípio de Inversão de Dependências

Esse princípio recomenda que uma classe cliente deve estabelecer dependências prioritariamente com abstrações e não com implementações concretas, pois abstrações (isto é, interfaces) são mais estáveis do que implementações concretas (isto é, classes). A ideia é então trocar (ou “inverter”) as dependências: em vez de depender de classes concretas, clientes devem depender de interfaces. Portanto, um nome mais intuitivo para o princípio seria **Prefira Interfaces a Classes**.

Exemplo:

```
class ControleRemoto {
    TVGenerica tv;
    ... // métodos
}

interface TVGenerica {
    ... // métodos de qualquer TV
}

class TVSamsung implements TVGenerica {
    ...
}
```

contra-exemplo:

```
class ControleRemoto {
    TVSamsung tv;
    ... // métodos
}

class TVSamsung {
    ...
}
```

D -

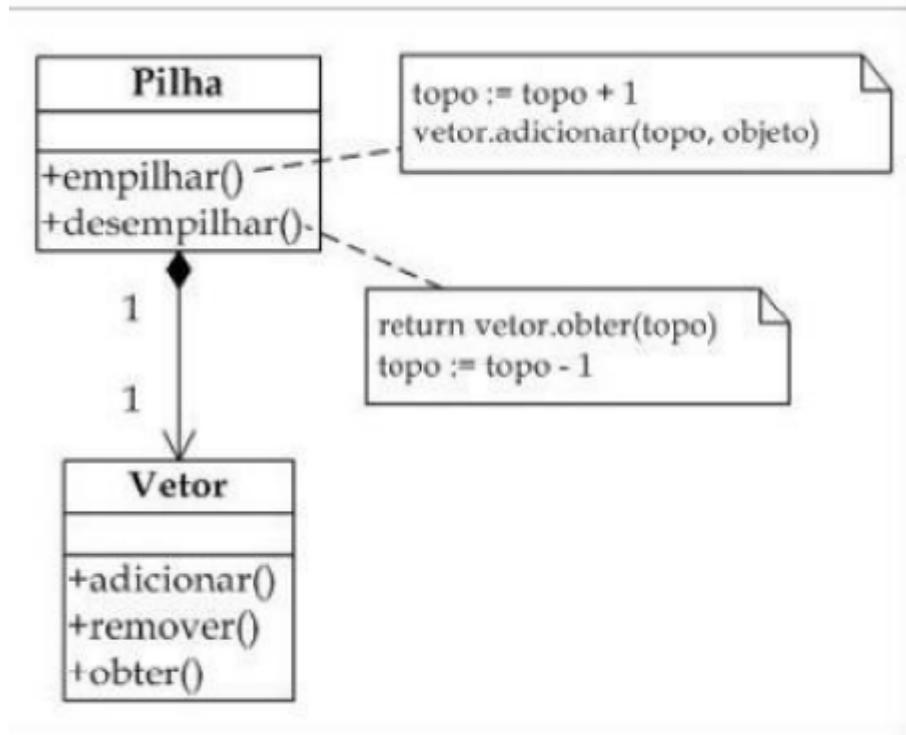
5.6.4 Prefira Composição a Herança

Antes de explicar o princípio, vamos esclarecer que existem dois tipos de herança:

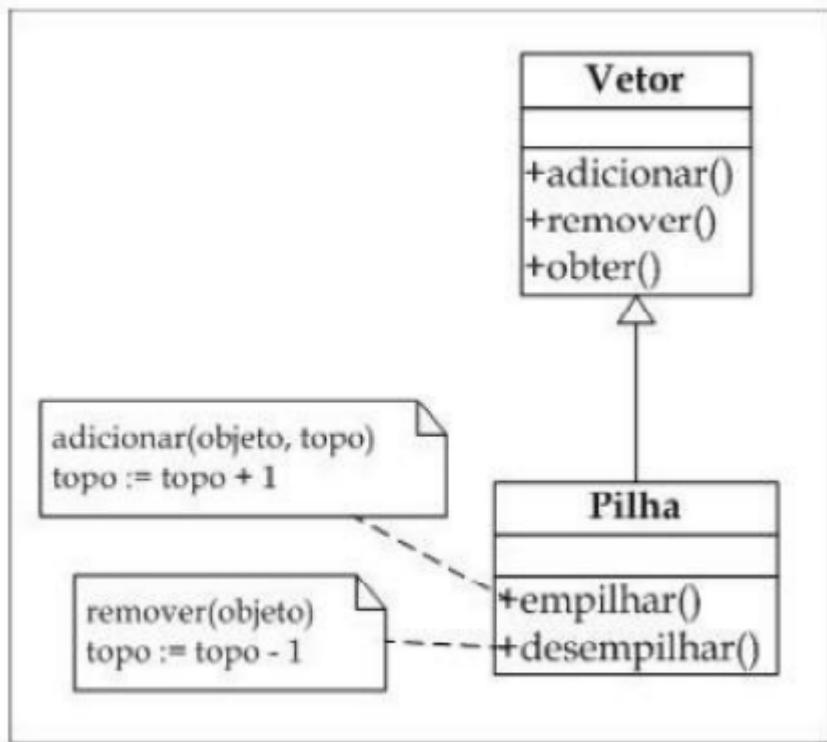
- **Herança de classes** (exemplo: `class A extends B`), que é aquela que envolve reuso de código. Não apenas neste capítulo, mas em todo o livro, quando mencionarmos apenas o termo herança estaremos nos referindo a herança de classes.
- **Herança de interfaces** (exemplo: `interface I extends J`), que não envolve reuso de código. Essa forma de herança é mais simples e não suscita preocupações. Quando precisarmos de nos referir a ela, iremos usar o nome completo: herança de interfaces.

Voltando ao princípio, quando orientação a objetos se tornou comum, na década de 80, houve um incentivo ao uso de herança. Acreditava-se que o conceito seria talvez uma bala de prata capaz de resolver os problemas de reuso de software. Argumentava-se que hierarquias de classes profundas, com vários níveis, seriam um indicativo de um bom projeto, no qual foi possível atingir elevados índices de reuso. No entanto, com o tempo, percebeu-se que herança não era a tal "bala de prata". Pelo contrário, herança tende a introduzir problemas na manutenção e evolução das classes de um sistema. Esses problemas têm sua origem no forte acoplamento que existe entre subclasses e superclasses, conforme descrito por Gamma e colegas no livro sobre padrões de projeto ([link](#)):

Exemplo:



Contra-exemplo:



E -

5.6.5 Princípio de Demeter

O nome desse princípio faz referência a um grupo de pesquisa da Northeastern University, em Boston, EUA. Esse grupo, chamado Demeter, desenvolvia pesquisas na área de modularização de software. No final da década de 80, em uma de suas pesquisas, o grupo enunciou um conjunto de regras para evitar problemas de encapsulamento em projeto de sistemas orientados a objetos, as quais ficaram conhecidas como Princípio ou Lei de Demeter.

O Princípio de Demeter — também chamado de **Princípio do Menor Privilégio** (*Principle of Least Privilege*) — defende que a implementação de um método deve invocar apenas os seguintes outros métodos:

- de sua própria classe (caso 1)
- de objetos passados como parâmetros (caso 2)
- de objetos criados pelo próprio método (caso 3)
- de atributos da classe do método (caso 4)

exemplo:

```
class PrincipioDemeter {  
    T1 attr;  
    void f1() {  
        ...  
    }  
  
    void m1(T2 p) { // método que segue Demeter  
        f1();           // caso 1: própria classe  
        p.f2();         // caso 2: parâmetro  
        new T3().f3(); // caso 3: criado pelo método  
        attr.f4();      // caso 4: atributo da classe  
    }  
}
```

Define quais chamadas de métodos são "permitidas" no corpo de um método



contra-exemplo:



```
void m2(T4 p) { // método que viola Demeter  
    p.getX().getY().getZ().doSomething();  
}
```

F -

5.6.6 Princípio Aberto/Fechado

Em resumo, o Princípio Aberto/Fechado tem como objetivo a construção de classes flexíveis e extensíveis, capazes de se adaptarem a diversos cenários de uso, sem modificações no seu código fonte.

exemplo:

Solução:

```
Comparator<String> comparador = new Comparator<String>() { ✓  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
};  
Collections.sort(nomes, comparador);
```

contra-exemplo:

```
List<String> nomes;  
nomes = Arrays.asList("joao", "maria", "alexandre", "ze");  
Collections.sort(nomes);
```

```
System.out.println(nomes);  
// resultado: ["alexandre", "joao", "maria", "ze"]
```

G -

5.6.7 Princípio de Substituição de Liskov

Em uma herança entre classes, quando as filhas sobrescrevem os serviços da classe pai, devem fazer a sobreposição com serviços melhores ou no mínimo iguais aos do pai. Nunca podem oferecer um serviço pior.

Exemplo:

```
class ControleRemoto {  
    // alcance de 10 m  
}  
  
class ControleRemotoPremium extends ControleRemoto {  
    // alcance de 20 m  
}
```



contra-exemplo:

```
class ControleRemoto {  
    // alcance de 10 m  
}  
  
class ControleRemotoXYZ extends ControleRemoto {  
    // alcance de 5 m  
}
```

