Fundação CECIERJ - Vice Presidência de Educação Superior a Distância

Curso de Tecnologia em Sistemas de Computação
Disciplina de Arquitetura e Projeto de Sistemas II
Gabarito – AD1 1° semestre de 2009.

Nome –

Observações:

1. Prova com consulta.

Atenção: Como a avaliação à distância é individual, caso sejam constatadas semelhanças entre provas de alunos distintos, será atribuída a nota ZERO a TODAS as provas envolvidas. As soluções para as questões podem ser buscadas por grupos de alunos, mas a redação final de cada prova tem que ser distinta. ALÉM DISSO, às questões desta AD respondidas de maneira muito semelhantes às respostas oriundas dos gabaritos já publicados de ADs de períodos anteriores, será atribuída a nota ZERO, incluindo também cópias diretas e sem sentido de tópicos dos slides das aulas.

Questão 1 [1 ponto]

Conceitue Processo Unificado e explique cada um de seus aspectos constituintes, ressaltando a ligação de cada um deles com sua importância prática do ponto de vista do cliente.

Resposta:

Processo Unificado é um processo de software que define um conjunto de atividades necessárias para transformar os requisitos do usuário em um sistema de software, caracterizado por ser dirigido a casos de uso, centrado na arquitetura e interativo e incremental. Os três aspectos a serem considerados neste processo de software são o desenvolvimento iterativo, o desenvolvimento evolutivo e o desenvolvimento ágil.

O desenvolvimento iterativo consiste em organizar o desenvolvimento em "mini-projetos" – um a cada interação –, de duração curta e fixa, com atividades de análise, projeto, programação e testes e cujo produto de cada iteração é um software parcial. Possibilita, dessa forma, a verificação constante pelo cliente dos requisitos implementados, com aumento de conhecimento sobre o software e redução de incertezas perante mudanças.

No desenvolvimento evolutivo, as especificações evoluem a cada iteração, com a construção de uma parte de software, de forma que o conhecimento sobre o software aumente. Diante de um cenário de mudanças (requisitos, ambiente e pessoas), a abordagem evolutiva tenta evitar correções, retrabalho e implementações de especificações "desnecessárias" do cliente (requisitos que não são realmente úteis ao software).

Já o desenvolvimento ágil fundamenta-se em respostas rápidas e flexíveis a mudanças, com replanejamento contínuo do projeto e entregas incrementais e constantes do software (refletindo tais mudanças). Em relação ao cliente, basta verificar que um dos pontos do manifesto ágil é: "colaboração do cliente vem antes de negociação de contrato".

Questão 2 [3 pontos]

A partir do Diagrama de Casos de Uso apresentado na Figura 1, correspondente ao Sistema do Domínio de Hotelaria, responda:

a) [1 ponto] Escolha um dos caso de uso apresentados e realize a sua descrição, conforme o *template* da Tabela 1;
b) [1 ponto] A partir do caso de uso escolhido, construa o(s) Diagrama(s) de Seqüência(s) correspondente(s);
c) [1 ponto] Construa o Modelo de Classes conceitual (artefato resultante da fase *Análise Orientada a Objetos*), mantendo-o coerente com as entidades utilizadas no Diagrama de Seqüência (b) e com as funcionalidades apresentadas no Diagrama de Casos de Uso (a).

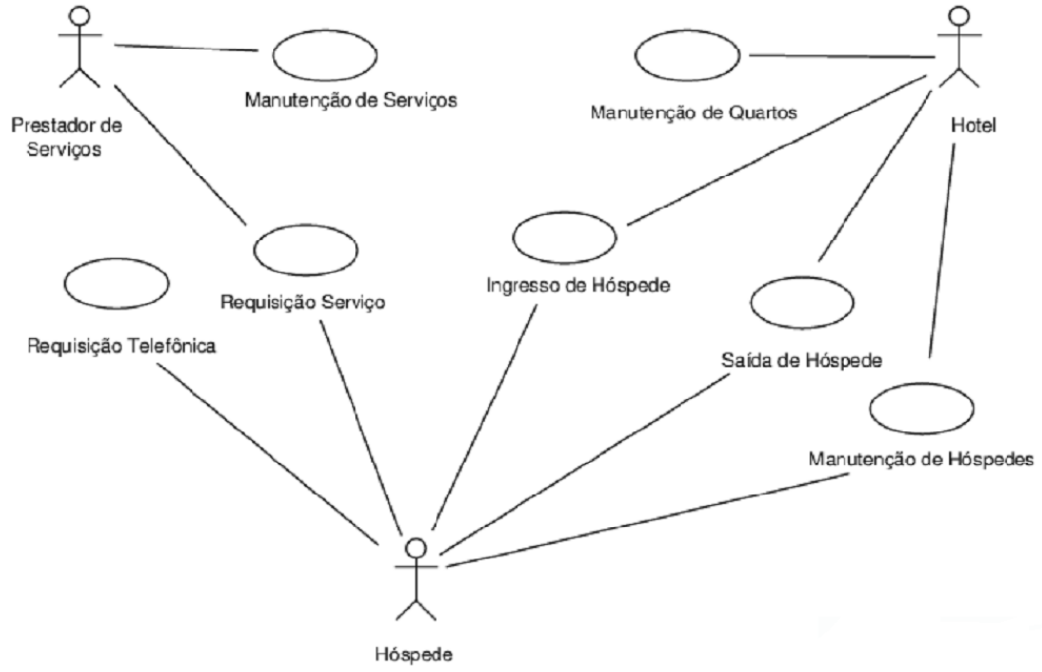


Figura 1 – Diagrama de Casos de Uso do Sistema do Domínio de Hotelaria

Tabela 1 – *Template* para Descrição de Casos de Uso

Nome:	<definir o nome do caso de uso>
Objetivo:	<descrever o objetivo do caso de uso>
Atores:	<descrever os atores que interagem com o caso de uso>
Pré-condições:	<descrever as pré-condições a serem atendidas para que o caso de uso possa ser executado>
***Trigger*:**	<definir que evento dispara a execução desse caso de uso>
Fluxo Principal:	<descrever o fluxo principal do caso de uso>
Fluxo Alternativo:	<descrever os fluxos alternativos do caso de uso, indicando que evento dispara cada um deles. Cada fluxo deve ser nomeado A1, A2 etc.>
Extensões:	<definir que extensões podem ser executadas>
Pós-condições:	<definir que produto ou resultado concreto o ator principal obterá ao final da execução do fluxo básico>
Regras de negócio:	<listar as regras de negócios que devem ser respeitadas na execução do caso de uso. Cada regra deve ser nomeada RN1, RN2 etc., e ser referenciada em algum fluxo do caso de uso (básico ou alternativo)>

Resposta:

a) Caso de uso escolhido: *Saída de Hóspede* (o aluno pode ter escolhido também qualquer um dos demais casos de uso: *Manutenção de Serviços*, *Requisição Telefônica*, *Requisição Serviço*, *Manutenção de Quartos*, *Ingresso de Hóspede* ou *Manutenção de Hóspedes*).

Nome:	Saída de Hóspede.
Objetivo:	O sistema deve permitir ao hóspede solicitar a sua saída (*check out*) do hotel, calculando o valor total a ser pago por sua estadia em aberto.
Atores:	*Hóspede* e *Hotel*.
Pré-condições:	A *data* e *hora* de saída do hóspede, referentes à estadia em aberto, são anteriores ou iguais à data e hora acordadas com o hotel durante a realização do caso de uso *Ingresso de Hóspede* para aquela estadia.
Trigger:	O funcionário do hotel aciona a opção *efetuar saída de hóspede*.
Fluxo Principal:	1. O funcionário do hotel aciona a opção *efetuar saída de hóspede*. 2. O sistema aciona o módulo “efetuar saída de hóspede” e solicita o nome ou CPF do hóspede em questão. 3. O funcionário preenche o campo “nome” com o nome do hóspede *ou* o campo “CPF” com o CPF do hóspede. 4. O sistema exibe os dados do hóspede e uma lista com o histórico de estadias, em ordem decrescente (da mais recente para a menos recente), sendo que a primeira estadia da lista corresponde àquela que está em aberto. [A1][A2] 5. O funcionário do hotel seleciona “estadia em aberto”. 6. O sistema verifica os dados da estadia em aberto do hóspede. [A3] 7. O sistema efetua o cálculo do preço a ser pago pelo hóspede, referente às despesas da estadia em aberto.

7.1. O sistema calcula o preço a ser pago pelo tipo do quarto ocupado durante a estadia do hóspede no hotel, a partir do número de diárias que compõem o período de estadia.
7.2. O sistema calcula o preço das chamadas telefônicas realizadas durante a estadia do hóspede no hotel, se houver, a partir do somatório dos preços calculados para cada uma das chamadas telefônicas registradas durante a realização do caso de uso de uso *Requisição Telefônica*.
7.3. O sistema calcula o preço dos pedidos de serviços solicitados durante a estadia do hóspede no hotel, se houve, a partir do somatório dos preços calculados para cada um dos pedidos de serviço registrados durante a realização do caso de uso *Requisição Serviço*.
7.4. O sistema calcula o preço total a ser pago pelo hóspede, a partir do somatório dos preços calculados nos itens 7.1, 7.2 e 7.3.

8. O sistema exibe um relatório sobre a estadia em aberto do hóspede, o qual inclui uma lista de itens: (i) número de diárias e preço correspondente; (ii) lista de chamadas telefônicas, contendo data, hora, tipo, duração, telefone de destino e preço correspondente de cada uma delas; e (iii) lista de pedidos de serviço, contendo nome, valor, quantidade, data, hora, nome do prestador e preço (incluindo a comissão do prestador) correspondente de cada um deles.
9. O hóspede efetua o pagamento em dinheiro (à vista), em cheque (à vista) ou com cartão (debito ou crédito).
10. O funcionário do hotel finaliza a estadia em aberto do hóspede, atualizando o estado do hóspede no sistema.

Fluxo Alternativo:

[A1] Nome de hóspede incorreto.
1. O sistema exibe a mensagem “nome de hóspede incorreto”. Volta ao passo 3 do fluxo principal para repetir o processo.

[A2] CPF de hóspede incorreto.
1. O sistema exibe a mensagem “CPF de hóspede incorreto”. Volta ao passo 3 do fluxo principal para repetir o processo.

[A3] Data e hora de saída de hóspede posterior à data e à hora acordadas durante o ingresso do hóspede no hotel.
1. O sistema exibe a mensagem de alerta “Data e hora de saída de hóspede posteriores à data e à hora acordadas durante o ingresso do hóspede no hotel”
2. O funcionário do hotel comunica ao hóspede.

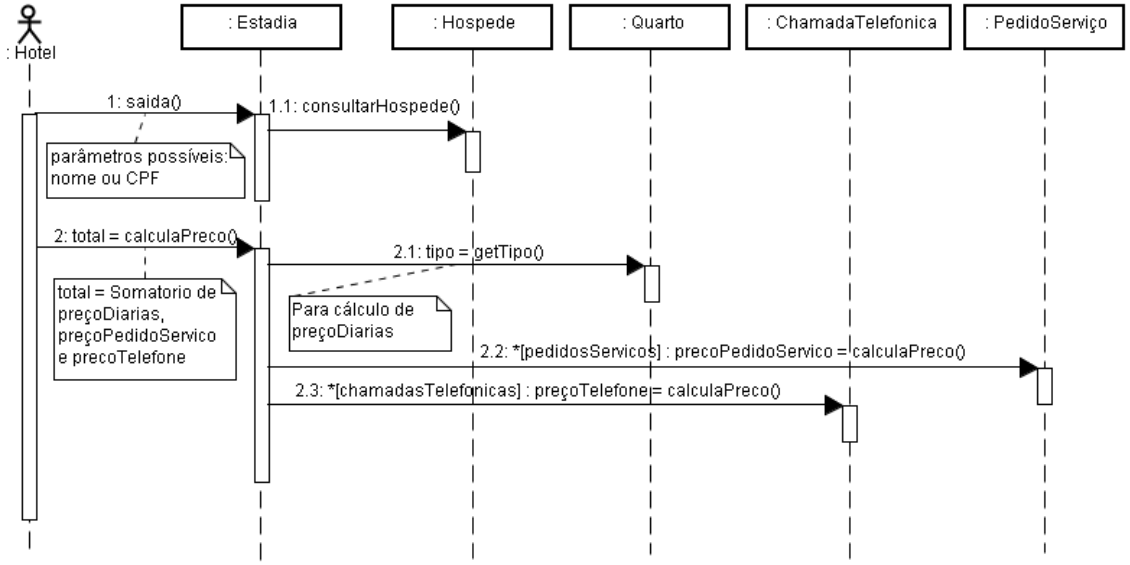
Caso o hóspede não esteja de acordo, solicitar a presença do hotel gerente para resolver a situação.
3. Voltar ao passo 7 do fluxo principal para continuar o processo.

Extensões: ---

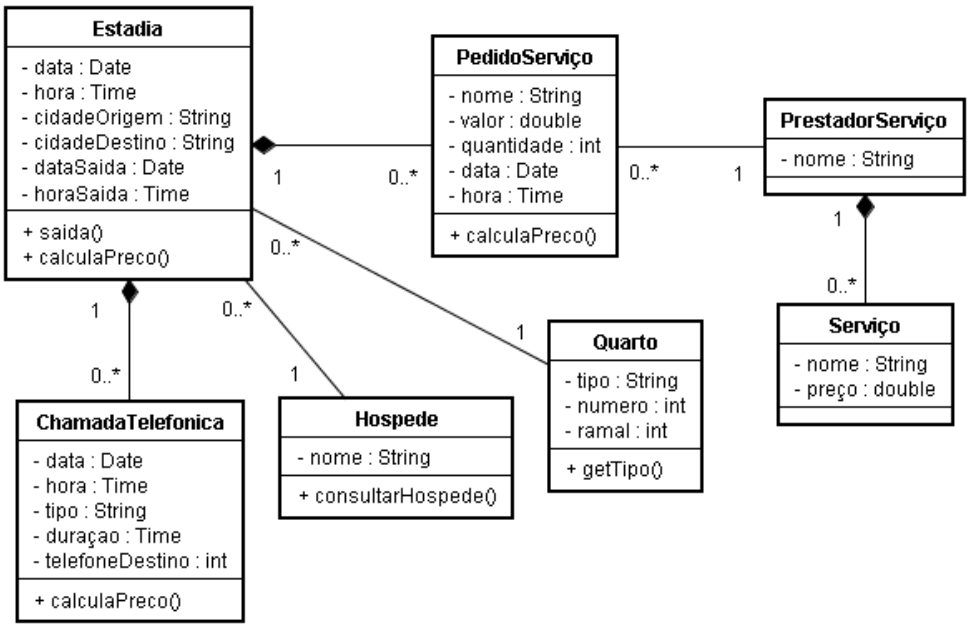
Pós-condições: A lista com o histórico de estadias do hóspede em questão não apresenta estadias em aberto (isso se propaga para as despesas relativas a requisições telefônicas e requisições de serviços). O quarto alocado para o hóspede é liberado para ser alocado para outra estadia.

Regras de negócio: RN01 – O cálculo do preço a ser pago durante a saída do hóspede, referente à sua estadia em aberto, é calculado a partir das despesas referentes à estadia (diárias), chamadas telefônicas e pedidos de serviços.

b) Diagrama de Seqüência para o caso de uso *Saída de Hóspede*. As mensagens 2.2 e 2.3 são referentes ao cálculo dos preços de todas as chamadas telefônicas e pedidos de serviço, respectivamente, realizados durante a estadia do hóspede no hotel, se houver. Os diagramas de seqüência e/ou mensagens correspondentes aos fluxos alternativos não foram representados.



c) Modelo de Classe conceitual:



Questão 3 [2 pontos]

Quais dimensões de estado abaixo você utilizaria para representar a classe `Data` (Figura 2)? Quais são as vantagens e desvantagens da sua escolha? Existe alguma outra solução

que contorna as desvantagens encontradas? Qual o espaço-estado de cada dimensão? Você consegue detectar alguma invariante para a classe `Data`?

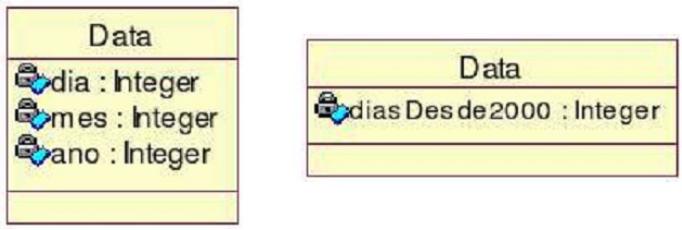


Figura 2 – Classe `Data`

Resposta:

Apesar de não existir "a resposta correta", para representar a classe `Data`, uma possível escolha de dimensões de estado seria a segunda opção, com o atributo *diasDesde2000* do tipo inteiro, pelo qual uma determinada data é descrita em termos da quantidade de dias antes (número negativo) ou depois (número positivo) do início do ano 2000.

Uma vantagem estaria no fato de que, na segunda opção, a representação dos dados permite mais facilmente calcular o intervalo de tempo entre duas datas, sendo de melhor manipulação em cálculos, computacionalmente falando. Por outro lado, na primeira opção, existem três atributos para representar uma data e um deles, *dia*, apresenta comportamento variado de acordo com o mês do ano (e.g., o mês de fevereiro tem 28 ou 29 dias, o mês de junho não tem 31 dias etc.) e de acordo com o tipo de ano (se é bissexto ou não). Ou seja, caso ocorra uma alteração em algum dos atributos, sem um devido cuidado, a classe pode deixar de representar uma data, ao se romper as suas propriedades.

A desvantagem da segunda opção estaria na compreensão natural das propriedades de uma data, pois o atributo apresenta uma quantidade de dígitos variável para representar uma data. Por outro lado, a representação de uma data em termos de suas partes constituintes (dia, mês e ano), na primeira opção, é de melhor manipulação humana, ao manter uma quantidade fixa de dígitos (dia e mês com 2 dígitos e ano com 4 dígitos) em cada atributo, favorecendo sua semântica.

Uma solução para contornar a desvantagem existente na segunda opção estaria na construção de uma interface semelhante à primeira opção, por meio da criação de métodos de acesso tais como *getDia()*, *getMes()* e *getAno()*, que extrairiam informações a partir de *diasDesde2000*, permitindo a utilização da classe `Data` mais facilmente.

Para a nova solução, o espaço-estado da dimensão *diasDesde2000* ficaria: *diasDesde2000* = [-∞..∞], onde ∞ depende da capacidade do tipo de dado

Por fim, podemos citar algumas invariantes detectadas, relacionadas às variações na quantidade de dias em cada mês do ano, além do caso relativo ao ano bissexto:

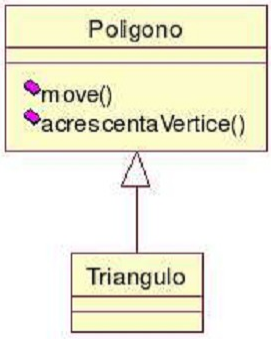
- "O mês de fevereiro tem 28 dias, exceto em ano bissexto, quando tem 29 dias"

- (getMes() = 2) AND (((getDia() = 29) AND (getAno() MOD 4 = 0)) OR (getDia() ≤ 28))

- "Os meses de janeiro, março, maio, julho, agosto, outubro e dezembro tem 31 dias"
  - ((getMes() = 1) OR (getMes() = 3) OR (getMes() = 5) OR (getMes() = 7) OR (getMes() = 8) OR (getMes() = 10) OR (getMes() = 12)) AND (getDia() = 31)
- "Os meses de abril, junho, setembro, novembro tem 30 dias"
  - ((getMes() = 4) OR (getMes() = 6) OR (getMes() = 9) OR (getMes() = 11)) AND (getDia() = 30)

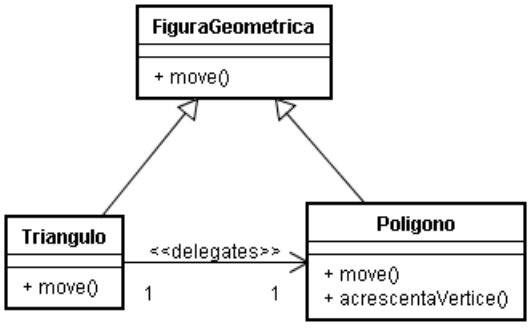
Questão 4 [1 ponto]

Qual modificação na estrutura abaixo poderia ser feita para possibilitar a manutenção do comportamento fechado global?



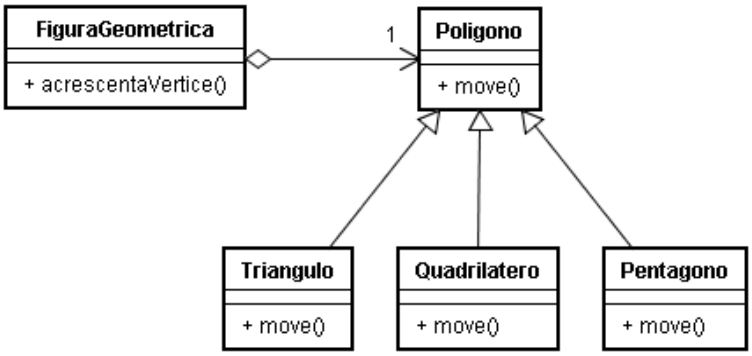
Resposta:

Existem alguns possíveis caminhos visando possibilitar a manutenção do comportamento fechado global: (i) fazer polimorfismo sobre o método `acrescentaVertice()`, com lançamento de exceção, o que representa uma solução ruim; (ii) evitar a herança de `acrescentaVertice()`, alterando a estrutura hierárquica; e (iii) preparar o projeto para possível reclassificação do objeto da classe `Triangulo` para outra classe (e.g., `Quadrilatero`, `Pentagono` etc.). Por exemplo, a solução exibida a seguir é relativa ao caminho (ii).



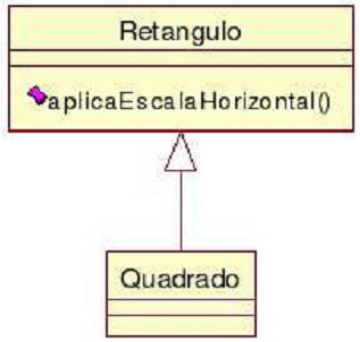
A solução exibida abaixo corresponde ao caminho (iii). Um objeto da classe `FiguraGeometrica` está associado a um objeto de alguma subclasse de *`Poligono`*. A depender da execução do método `acrescentaVertice()` de

`FiguraGeometrica`, o objeto associado ao objeto destas pode ter suas propriedades alteradas, devendo-se instanciar um outro objeto de uma subclasse mais adequada de *`Poligono`*. Ou seja, deve-se permitir que um objeto modifique o seu comportamento em função do seu estado interno, gerando o efeito de uma troca de tipo de objeto em tempo de execução. Isso equivale a criar um novo polígono, mantendo as características anteriores (antigos vértices) e adicionando uma nova característica (novo vértice).



Questão 5 [1 ponto]

Supondo que a hierarquia abaixo não possa ser modificada, como você faria para manter o comportamento fechado de `aplicaEscalaHorizontal()` em relação a classe `Quadrado` (que você tem acesso ao código) sem utilizar lançamento de exceções, visto que aplicar escala horizontal em um quadrado o transforma em um retângulo?



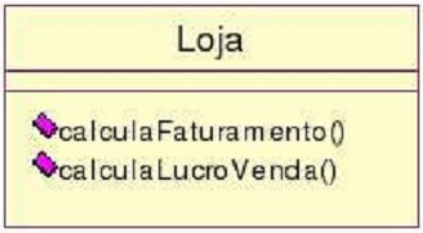
Resposta:

Para manter o comportamento fechado de `aplicaEscalaHorizontal()`, deve-se levar em consideração que (i) a hierarquia não pode ser modificada, (ii) só é permitido o acesso ao código da classe `Quadrado`, (iii) `aplicaEscalaHorizontal()` transforma um quadrado em um retângulo e (iv) não é possível utilizar lançamento de exceção. A partir dessas premissas, uma possível solução consiste em aplicar escala vertical na exata proporção em que a escala horizontal foi aplicada, reimplementando `aplicaEscalaHorizontal()` em `Quadrado` por invocar o correspondente método da superclasse e, em seguida, realizar uma chamada ao método

`aplicaEscalaVertical()` definido em `Quadrado`, mantendo as propriedades e características reais de um quadrado.

Questão 6 [1 ponto]

Suponha que a classe `Loja` foi construída inicialmente com o método `calculaFaturamento()`, que soma o lucro de todas as vendas. Posteriormente, foi criado o método `calculaLucroVenda()`, que calcula o lucro de uma única venda. Você considera esta situação um caso de interface com comportamento replicado? Se sim, o que deve ser feito?

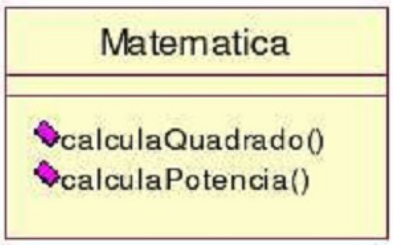


Resposta:

Sim, isso representa um caso de interface com comportamento replicado, pois ambos os métodos calculam lucros de vendas, quer seja de todas as vendas, quer seja de uma venda em particular, além de aumentar o acoplamento ao obrigar o cliente a saber quais as vendas que ele deseja incluir no faturamento, por exemplo. Além disso, em um bom Projeto Orientado a Objetos, o método `calculaLucroVenda()` deveria pertencer à classe Venda, visando manter o encapsulamento e não aumentar o acoplamento, e o método `calculaFaturamento()` deveria apenas somar os lucros de todas as vendas, lucros estes calculados individualmente por cada um dos objetos de `Venda`. Dessa forma, uma possível solução consiste em refatorar esses métodos, eliminando-os, a fim de criar um método `calculaLucroVendas(dataInicial, dataFinal)` na classe `Loja`. Este método recebe como argumentos a data de início e a data de fim da fatura a ser calculada, de forma que seu funcionamento se baseie invocar um objeto apropriado (do tipo `CatalogoVendas`) que recupere as vendas de um certo período e aciona o método `calculaLucroVenda()` (da classe `Venda`) para calcular o lucro de cada venda, somando-as iterativamente. Assim, melhora-se o encapsulamento, além de manter a funcionalidade desejada da classe `Loja`.

Questão 7 [1 ponto]

Segue, abaixo, um caso claro de interface com comportamento duplicado, onde o método `calculaQuadrado()` fornece o valor de um número ao quadrado, e o método `calculaPotencia()` fornece o valor de um número elevado a outro número. Quais medidas podem ser tomadas a curto, médio e longo prazo para que o primeiro método possa ser banido da classe?



Resposta:

A idéia para resolver este problema consiste na remoção do método `calculaQuadrado()` da classe `Matemática`, visando remover o comportamento duplicado identificado. No entanto, esse processo deve ser gradativo. Em curto prazo, como `calculaQuadrado()` é um serviço público de `Matematica`, este método deve ser codificado em termos de `calculaPotencia()` e os novos clientes devem ser avisados, para que não utilizem `calculaQuadrado()`. Em médio prazo, torna-se necessário comunicar aos possíveis usuários deste serviço que uma modificação precisa ser feita na API (*Application Programming Interface*) de `Matematica`. Nesse período, tanto o método `calculaQuadrado()` como o método `calculaPotencia()` devem estar disponíveis, para possibilitar uma migração gradativa dos clientes. Em longo prazo, somente quando o período de migração se encerrar, é que a versão antiga do serviço (`calculaQuadrado()`) pode ser removida, restando somente a nova versão do mesmo.