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 – AD2 2° semestre de 2007.

Nome –

Observações:

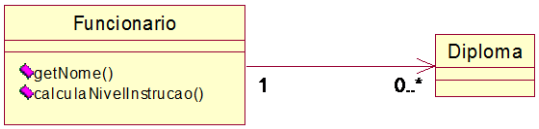
1. Prova com consulta.

Atenção: Como a avaliação à distância é individual, caso seja constatado semelhanças entre provas de alunos distintos, a todas será atribuída a nota ZERO. As soluções para as questões podem sim, serem buscadas por grupos de alunos, mas a redação final de cada prova tem que ser individual.

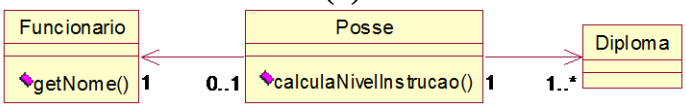
Questão 1 [1 ponto]

Cenário do problema: O sistema de controle de treinamentos de uma empresa precisa identificar, para cada funcionário da empresa, toda a sua formação até aquele momento, para poder lhe oferecer cursos do seu interesse e condizentes com o seu nível de instrução. Para isso, existe um método chamado calculaNivelInstrucao() que fornece o nível de instrução de um objeto da classe Funcionario (que é utilizada em outros sistemas da empresa) em função de objetos da classe Diploma, que representa os diplomas ou certificados obtidos pelo funcionário durante a sua carreira profissional. Analise os 4 modelos abaixo, citando para cada modelo as suas vantagens e desvantagens.

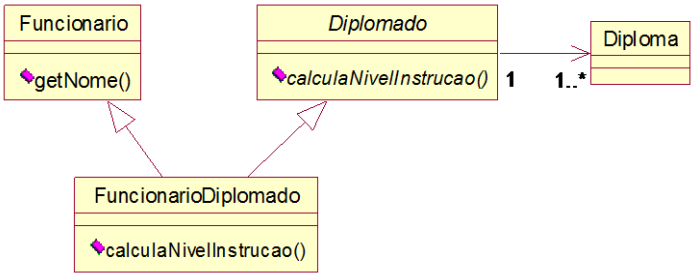
(a)



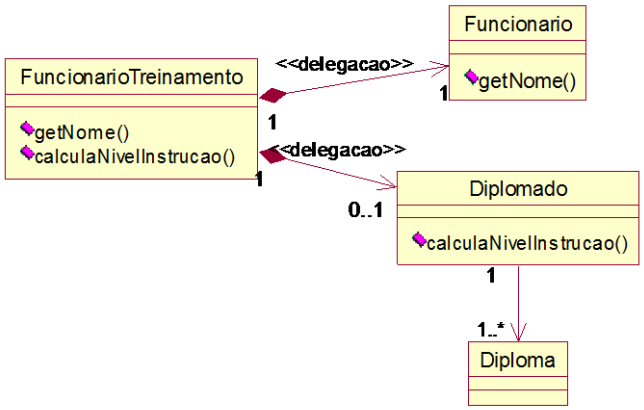
(b)



(c)



(d)



R: (a) Este modelo é bem simples e de fácil compreensão. Entretanto, insere-se na classe `Funcionario` (utilizada em outros sistemas da empresa) o método `calculaNivelInstrucao()`, que será utilizado apenas pelo sistema de controle de treinamentos, provocando acoplamento entre `Funcionario` e `Diploma` (uma classe do sistema controle de treinamentos). Dessa forma, deve-se tentar desacoplar decisões locais de decisões globais a fim de evitar comportamento intrusivo no sistema e deixar suas classes mais coesas.

(b) Neste modelo, buscou-se evitar o relacionamento direto entre as classes `Funcionario` e `Diploma` através da classe `Posse`, melhorando a questão do acoplamento indevido. Entretanto, não é possível, a partir da classe `Funcionario` obter a sua lista de diplomas `Diplomas`, pois não existe navegabilidade de `Funcionario` para `Posse`.

(c) O modelo apresentado melhora a situação de acoplamento e de relacionamento entre `Funcionario` e `Diploma` ao criar a classe `Diplomado`, que se relaciona com a classe `Diploma` por meio do método `calculaNivelInstrucao()`. A subclasse `FuncionarioDiplomado` constitui, nesse caso, o elo que contém funcionários que são diplomados. Um possível problema consiste na herança múltipla, quando ocorrer a tradução desse modelo para o modelo de implementação, devendo-se ocorrer ajustes dada a linguagem de programação escolhida (como em Java, por exemplo).

(d) O quarto modelo oferece as melhores características de projeto OO, uma vez que é criada uma classe `FuncionarioTreinamento` que representa um perfil (papel), composto pelas classes `Funcionario` e `Diplomado`, evitando a herança múltipla. A herança permite o compartilhamento baseado em classes, enquanto que a delegação

permite o compartilhamento baseado em objetos. Dessa forma, este mecanismo permite o repasse da mensagem para o outro objeto, uma vez que o objeto "delegador" do serviço, no caso aqueles da classe `FuncionarioTreinamento`, contém uma referência para o objeto responsável pela execução, no caso aqueles das classes `Funcionario` e `Diploma`, podendo obter resultados pelos dois métodos que a nova classe referencia. Um aspecto negativo desse modelo é o aumento de complexidade da solução. Contudo, essa complexidade extra pode ser compensada pelos ganhos de uma solução mais coesa e menos acoplada.

Questão 2 [0,5 ponto]

Explique o que são heurísticas de projeto e para que servem.

R: Heurísticas de projeto representam o resultado da explicitação do conhecimento de vários projetistas experientes, ou seja, orientações que apóiam projetistas na tomada de decisões na etapa de projeto de software. Entretanto, as heurísticas não pretendem substituir metodologias e sim auxiliá-las a gerar bons projetos de software ao motivarem a aplicação de determinado padrão de projeto – elas são úteis na maior parte dos casos, mas não em todos, pois pode haver conflitos mútuos entre algumas delas.

Questão 3 [0,5 ponto]

Conceitue reutilização de software e cite alguns de seus benefícios e dificuldades.

R: Reutilização de software é o processo de incorporar em um novo produto artefatos de software preexistentes, ou de uma forma geral o conhecimento sobre o desenvolvimento destes artefatos. Alguns de seus benefícios esperados são: (i) melhores índices de produtividade; (ii) produtos de melhor qualidade, mais confiáveis, consistentes e padronizados; (iii) redução dos custos e tempo envolvidos no desenvolvimento de software; e (iv) maior flexibilidade na estrutura do software produzido, facilitando sua manutenção e evolução. Por outro lado, existem algumas dificuldades: (i) identificação, recuperação e modificação de artefatos reutilizáveis; (ii) compreensão dos artefatos recuperados; (iii) garantia da qualidade dos artefatos reutilizáveis; (iv) composição de aplicações a partir de componentes; (v) barreiras psicológicas, legais e econômicas; e (vi) a necessidade da criação de incentivos à reutilização.

Questão 4 [0,5 ponto]

Explique o que são *frameworks* e discorra sobre seus tipos.

R: *Frameworks* (arcabouços) correspondem a projetos ou arquiteturas de alto nível, compostos por classes especialmente projetadas para serem refinadas e utilizadas em conjunto. São estruturas de suporte definidas nas quais outros projetos de software podem ser organizados e desenvolvidos. São projetados com a intenção de facilitar o desenvolvimento de software, habilitando projetistas e programadores a gastarem mais tempo determinando as exigências do software do que com detalhes de baixo nível do

sistema. Algumas características de um *framework* são: (i) provê uma solução para uma família de problemas semelhantes; (ii) utiliza um conjunto de classes e interfaces que mostra como decompor a família de problemas; (iii) mostra como objetos dessas classes colaboram para cumprir suas responsabilidades; e (iv) por fim, o conjunto de classes deve ser flexível e extensível para permitir a construção de várias aplicações com pouco esforço, especificando apenas as particularidades de cada aplicação.

Existem *frameworks* caixa-branca e caixa-preta. Nos *frameworks* caixa-branca o comportamento específico da aplicação é inserido na arquitetura genérica, somando-se métodos às subclasses de uma ou mais classes do *framework*. Cada método somado a uma subclasse deve estar de acordo com as convenções da respectiva superclasse. Os *frameworks* que fazem uso deste processo de especialização obrigam quem os utiliza a conhecer os seus dispositivos internos. Por outro lado, uma segunda forma de especialização é representada pelos *frameworks* caixa-preta: o *framework* recebe um conjunto de parâmetros previamente estabelecidos que representa o comportamento específico da aplicação. Este fornecimento de parâmetros é feito por protocolo, através da interface de cada classe do *framework* e, por isso, esta forma de especialização requer que seja conhecida apenas esta interface.

Questão 5 [0,5 ponto]

O que são padrões de software? Qual a sua relação com reutilização de software?

R: Padrões de software (*Design Patterns*) descrevem soluções para problemas recorrentes no desenvolvimento de sistemas de software orientado a objetos. Estes problemas ocorrem repetidas vezes em nosso meio e incluem soluções genéricas de tal forma que podem ser utilizadas de várias maneiras não idênticas – um padrão de projeto estabelece um nome e define o problema, a solução, quando aplicar esta solução e suas conseqüências. Visam facilitar a reutilização de soluções de projeto de software, além de acarretarem em um vocabulário comum de projeto, facilitando a comunicação, documentação e aprendizado dos sistemas de software. Isto é, a incorporação do conhecimento sobre o processo de desenvolvimento é a maior riqueza que os padrões trazem para este contexto.

Questão 6 [1 ponto]

Escolha dois padrões GRASP e dois padrões GoF e explique-os.

R:
Padrões GRASP:

- *Creator*: o problema refere-se a entender quem deveria ser responsável pela criação de uma nova instância de uma classe. A criação de objetos é uma atividade comum em sistemas orientados a objetos, sendo necessário um princípio que atribua responsabilidades para essa criação. A solução adotada consiste em atribuir à classe A a responsabilidade de criar instâncias da classe B se ao menos:

(i) A contém objetos de B; (ii) A agrega objetos de B; (iii) A registra objetos de B; (iv) A usa de maneira muito próxima objetos de B; ou (v) A tem os dados necessários para a construção de objetos de B. Por exemplo, em um sistema de ponto de venda (PDV), a classe `Pedido` deveria ser responsável por criar instâncias da classe `ItemPedido`. Dentre os benefícios, está o fato de não aumentar o acoplamento, pois a visibilidade entre as classes envolvidas já existia.

- *Controller*: neste caso, o problema relaciona-se a descobrir quem deveria ser responsável por tratar um evento de sistema. Os eventos de sistema estão associados às mensagens de sistema, que são geradas a partir dos passos dos casos de uso. A solução adotada consiste em tratar os eventos de sistemas por uma das seguintes classes: representante do sistema como um todo (*Facade*), representante do negócio ou da organização (objeto raiz) ou representante do caso de uso em questão (tratador artificial). Por exemplo, em um sistema de ponto de venda (PDV), o evento de registro de um item de pedido pode ser executado por uma classe `PDV` (representando o sistema como um todo), uma classe `Loja` (representando o negócio) ou uma classe `ControleCompraItens` (representando o caso de uso de compra de itens). Os benefícios deste padrão são a separação *view-controller* facilitando a reutilização de componentes específicos de negócio e permitindo o uso dos serviços através de processamento em lote (*batch*).

Padrões GoF:

- *Singleton*: é um padrão de criação que garante a existência de apenas uma instância de uma classe, mantendo um ponto global de acesso a ela. A principal razão para este padrão é que em quase todo tipo de sistema existem classes com uma única instância. Por exemplo, uma aplicação que precisa de uma infra-estrutura de *log* pode utilizar uma classe no padrão *singleton*, onde existe apenas um objeto responsável pelo *log* em toda a aplicação, que é acessível unicamente através da classe *singleton*.

- *Composite*: é um padrão de estrutura utilizado para representar um objeto que é constituído pela composição de objetos similares a ele, ou seja, compor objetos utilizando uma estrutura de árvore para representar hierarquias de *todo-parte*. Neste padrão, o objeto composto possui um conjunto de outros objetos que estão na mesma hierarquia de classes a que ele pertence. Além disso, esta forma de representar elementos compostos em uma hierarquia de classes permite que os objetos do tipo *todo* ou do tipo *parte* sejam tratados da mesma maneira. Assim, todos os métodos comuns às classes que representam objetos atômicos da hierarquia poderão ser aplicáveis também ao conjunto de objetos agrupados no objeto composto. Este padrão é normalmente utilizado para representar listas recorrentes – ou recursivas – de elementos. Por exemplo, no projeto de um sistema de arquivos, espera-se que tanto um arquivo quanto um diretório possam informar o seu tamanho.

Questão 7 [1 ponto]

a) Defina componentes, explicando suas características fundamentais.
b) Qual a diferença entre um processo convencional e um processo ideal com reutilização e DBC?

R: (a) Componentes são elementos que podem ser desenvolvidos independentemente e entregues como unidades, que podem ser compostos com outros componentes, e que explicitam as suas funcionalidades por meio de interfaces providas e requeridas. Suas características fundamentais são: (i) unidade de composição auto-contida (encapsulamento), reunindo um conjunto de características e operações implementadas da forma mais genérica possível; (ii) interfaces bem definidas (contrato), provendo e requerendo informações do mundo exterior por meio de um protocolo de comunicação bem definido; (iii) grau de maturidade e documentação (qualidade), contendo um conjunto de artefatos que o descrevem a fim de reduzir riscos devido ao seu propósito de reutilização; (iv) especificado em diferentes níveis (abstração), abrangendo não apenas artefatos de código, mas também artefatos em nível de análise, projeto e testes; e (v) obedece a restrições (arquitetura/plataforma) previamente identificadas e documentadas.

(b) A diferença consiste na inserção de uma fase a mais no processo tradicional (análise – projeto arquitetural – projeto detalhado – codificação – testes), entre projeto arquitetural e projeto detalhado: a busca por componentes. O objetivo é incorporar reutilização ao longo do processo de desenvolvimento por meio do desenvolvimento baseado em componentes. A busca de componentes consiste na customização de componentes e/ou na adaptação do projeto arquitetural, uma vez que nem sempre os componentes desejados são encontrados ou mesmo aqueles encontrados não são exatamente o que se esperava.

Questão 8 [1 ponto]

Defina arquitetura de software com suas próprias palavras. A partir disso, cite quais são as qualificações necessárias de um arquiteto de software.

R: Arquitetura de software é o estudo da organização global dos sistemas de software bem como do relacionamento entre seus subsistemas e componentes. Ela serve como uma estrutura que permite o entendimento de componentes de um sistema e seus inter-relacionamentos, especialmente daqueles atributos que são consistentes ao longo do tempo e de implementações. O termo também se refere à documentação da arquitetura de software do sistema, a qual facilita a comunicação entre os *stakeholders*, registra as decisões iniciais acerca do projeto de alto-nível e permite a reutilização do projeto dos componentes e padrões entre projetos. Além disso, arquitetura de software deve ser vista e descrita sob diferentes perspectivas e deve identificar seus componentes, relacionamento estático, interações dinâmicas, propriedades, características e restrições. A disciplina de arquitetura de software é centrada na idéia da redução da complexidade através da abstração e separação de interesses. Algumas questões são alvos da disciplina de arquitetura de software: (i) questões estruturais envolvem organização e estrutura geral

de controle; (ii) protocolos de comunicação e sincronização; (iii) atribuição de funcionalidade a componentes de projeto; (iv) escalabilidade e desempenho; e (v) seleção de alternativas de projeto. Nesse sentido, algumas qualificações necessárias de um arquiteto de software correspondem a: (i) conhecimento sobre domínio e tecnologias; (ii) entendimento de aspectos técnicos do desenvolvimento; (iii) técnicas de elicitação, modelagem e métodos; (iv) entendimento das estratégias de negócio da instituição; (v) conhecimento sobre produtos, processos e estratégias de concorrentes; e (vi) criatividade e bom uso de abstrações.

Questão 9 [1 ponto]

Baseado no capítulo 2 do livro do Mendes, enumere uma vantagem e uma desvantagem para cada um dos estilos arquiteturais estudados (*Pipes & Filters*, em camadas, processos distribuídos e orientado a objetos).

R:

Pipes & Filters

- Vantagens: simplicidade; e facilidades para compor e paralelizar o sistema;
- Desvantagens: baixo desempenho (abstração de dados primitiva e gerência de *buffers*); e difícil interatividade e cooperação entre filtros.

Em camadas

- Vantagens: flexibilidade; e reutilização de uma infra-estrutura de computação pré-existente;
- Desvantagens: determinação do número de camadas; e comprometimento do desempenho devido à comunicação entre as camadas.

Processos distribuídos

- Vantagem: decomposição do sistema em partes menores de modo a facilitar modificações necessárias;
- Desvantagens: restrições topológicas; sujeito a problemas de sincronização de mensagens.

Orientado a objetos

- Vantagens: permite a organização de programas; e ajuda a manter a integridade dos dados do sistema;
- Desvantagem: devido à granularidade fina dos objetos e a dependência entre eles, a manutenção e a reutilização podem ser dificultadas, pois para modificar ou reutilizar um objeto pode ser necessário conhecer e atuar sobre outros objetos do sistema.

Questão 10 [1 ponto]

Cite qual a contribuição de cada uma das diferentes visões requeridas pela arquitetura para o projeto de um sistema.

R:
Visão lógica: oferecer um retrato estático das classes fundamentais e seus relacionamentos.

Visão de desenvolvimento: mostrar como o código é organizado em pacotes e bibliotecas.

Visão de processos: demonstrar as atividades e tarefas realizadas pelo sistema.

Visão física: mostrar os processadores, dispositivos e ligações no ambiente de operação.

Visão de cenários: explicar como as quatro visões operam em conjunto.

Questão 11 [1 ponto]

Caracterize a diferença entre engenharia de domínio e engenharia de aplicação.

R: A engenharia de domínio se refere a um processo que visa identificar, representar e implementar o núcleo de artefatos reutilizáveis (*core assets*) de um domínio, ao passo que a engenharia de aplicação se utiliza desses artefatos reutilizáveis para a construção de produtos de software mais flexíveis e que possam evoluir mais facilmente.

Questão 12 [1 ponto]

Considerando arquiteturas *Web* responda:

a) Quais os seus componentes fundamentais?
b) Explique um canário típico de comunicação em um sistema *Web* que apresenta arquitetura em três camadas.
c) Selecione dois estilos arquiteturais para *Web* e explicite suas características, vantagens e limitações.

R: (a) HTML (páginas *web* estáticas), componentes que rodam no cliente (e.g., Applets e ActiveX), scripts que rodam no cliente (e.g., JavaScript), scripts que rodam no servidor (e.g., PHP, ASP, JSP), componentes que rodam no servidor (e.g., Servlets e EJB) e banco de dados.

(b) Inicialmente, o cliente no *browser* solicita um serviço ao servidor usualmente após o preenchimento de um formulário HTML. A seguir, o servidor interpreta a solicitação na camada de aplicação. Se necessário, a camada de aplicação se comunica com a camada de armazenamento para acessar dados. Assim, a camada de aplicação redireciona o fluxo para a camada de apresentação a fim de que o servidor construa uma página de resposta. Por fim, o servidor retorna a página de resposta.

(c) *Thin Client*: representa um estilo fundamentado na mínima utilização dos recursos do cliente, exigindo mínima capacidade de seu navegador. Algumas de suas

características são: (i) estilo mais apropriado para aplicações *Internet*; (ii) baixo controle da configuração do navegador cliente; e (iii) toda a lógica de negócio reside no servidor. Suas vantagens: (i) independência total de plataforma ou navegador; e (ii) concentração da lógica de negócio no servidor. Apesar disso, apresenta limitações: (i) interface com o usuário limitada pela linguagem HTML; e (ii) maior número de transações com o servidor.

AJAX: consiste em um estilo baseado na utilização de *scripts* de cliente de forma assíncrona, permitindo a carga parcial de páginas. Algumas de suas características são: (i) permite a carga parcial de dados em uma página e (ii) combina *JavaScript* e XML. Uma vantagem está na carga parcial e assíncrona de dados para o cliente, ao passo que uma limitação se refere à dependência (limitada) com o navegador.