Lab 4: Projeto OO com Design Patterns e bons princípios

Problema: Sistema de notas fiscais revisitado

Obedecer as restricoes legais e requisitos do product owner é critico. Ambos são expressos como requisitos funcionais e não-funcionais, e a nota consiste em convencer o professor que cada um dos requisitos estão satisfeitos de forma forte (não é possível que o programador-usuário quebre o requisito), elegante (sem excessivo mau cheiros), e usando DPs. Funcionar o demo é necessário mas não suficiente. Deve-se explicar porque o requisito é satisfeito usando DPs, e/ou principios como GRASP e SOLID.

Rationale:

Já fizeram projeto orientado a objeto, mas agora o proposito é aplicar os conceitos do curso, inclusive do 1o bimestre (acoplamento abstrato, diferenciação entre agragação/composição/associação, diagramas de classe) e principalmente DPs em um código arquitetural que atenda á uma série de requisitos.
Note que também espera-se que aproveite o horário da aula para discutir a solução antes de codificar. Espero que haja muita discussão e diagramação, refatoração, testes.[1] E espero que usem os conceitos do curso, especialmente DPs na explicação das suas soluções (conceito de DPs como linguagem)

Grupo: Trios de 3 indivíduos (não de 1, nem de 2, nem de 4)

Objetivo e nota:

nota final:

min(10 , 10.5 * <# requisitos *demonstrados* satisfeitos> / <# requisitos >)
Só satisfazer o requisito não vale nada, tem que explicar.

O professor não se auto-convencerá da satisfação dos requisitos procurando a solução no seu código nem entenderá o seu código sozinho. O professor é convencido que o requisito é satisfeito por, sempre que aplicável:

1. Um demo ou teste demostrando a utilização das classes implementadas no contexto do requisito, com o numero do requisito no nome do demo ou teste.
2. Um diagrama de classes e/ou sequência comentado e texto indicando quais e como DPs foram utilizados, e porque e como o requisito é satisfeito. É recomendado usar

[1] Recomendo que utilizem git ou outro controle de versão pra tudo, código e textos (escrevam e versionem partes, no final juntem um texto só). Diminui a preguiça de refatorar, organiza o trabalho dos 3, e evita catastrofes de perder ou bagunçar as versões.

o conceito de cada DP para explicar os conceitos do sistema, como explicamos o Publish/Subscribe usando o Observer. Também é possível utilizar conceitos como GRASP e SOLID para explicar porque a sua solução é elegante.

3. Testes indicando que tentativas de burlar os requisitos resultam em exceções, erros, etc., e não permitem execução normal.

Vale um item demonstrando vários requisitos se for indicado onde cada requisito esta demonstrado.

Regra: cada requisito deve ser demonstrado, mas a aplicação é uma só. Não vale entregar uma aplicação para cada requisito. A demonstração de requisito pode conter um demo ou teste que usa a aplicação.

Regra: deve-se implementar apenas os atributos que sejam relevantes para demonstrar os requisitos, implementar os DPs, no espírito do conceito de código arquitetural que usamos na P1. Atributos que sejam apenas "enchimento", podem ser representados por um inteiro ou string chamado "outros". Se precisar fazer buscas, uma busca pelo campo "outros" representaria busca complexas com ANDs e ORs em todos os campos agregados em "outros", sem precisar implementar.

Regra: Entrada de usuario e BDs devem ser Mocks, E devem estar separados do código de produção. Ou seja:
Quando for preciso armazenar e obter dados, deve haver uma classe responsavel por isso, simulando o BD.
Quando for preciso entrada de usuario humano, deve haver uma classe com a responsabilidade de gerar dados simulados, e não interromper o demo para pedir que o humano digite. Por exemplo, para criar um IV com um P/S, uma GUI geraria janelas para o humano digitar vários dados, depois listar, buscar, ordenar, e finalmente selecionar um P/S do BD. Mas o resultado dessa GUI para o sistema seria um conjunto de dados para um novo IV inclusive um ponteiro para um P/S retirado do BD. Um Mock pode substituir a GUI para gerar estes dados. Por exemplo, existindo 4 subclasses de P/S definidos, um Mock pode retornar uma delas aleatoriamente.

Regra: TODOS OS DEMOS E TESTES DEVEM EXECUTAR INSTANTANEAMENTE SEM PARADAS PARA ENTRADAS HUMANAS. MOCKS DEVEM SUSBTITUIR TODA ENTRADA DE DADOS.

Regra: não e obrigado a seguir exatamente o DP sugerido para um requisito. Algumas vezes inclusive a solução que imaginamos é parecida mas não exatamente igual ao DP sugerido. Mas sugerimos um DP exatamente porque pensamos que a idéia se aplica à solução. Vale copy-paste de código de exemplos de DPs, de sites ou livros.

Conceitos (não necessariamente classes):

1. Nota Fiscal (NF) contem itens de venda (IV).
2. ~~NF contem informação do cliente~~.
3. Item de venda (IV): Deve estar associado a um produto ou serviço pre-existente.

4. Produto/Serviço (P/S): produtos e servicos que podem ser vendidos.
5. Imposto: Define taxas para produtos e servicos. Produtos e servicos especificos podem ter taxas diferenciadas.
6. Validador de NF: depois de preenchida, a NF é validada: os impostos são calculados, um ID único é gerado, e depois disso a NF é imutável.
7. ~~Cliente: dados de um cliente. contem CPF valido~~.
8. ~~Cadastro: Acessa BD de clientes, possui metodos para encontrar clientes e para cadastrar novo cliente. Nao deve cadastrar cliente se o CPF for invalido.~~
9. ~~VerificadorCPF: algoritmo que verifica se um CPF eh valido~~.

Restrições e Requisitos

1. Restrição legal: NF não pode ter zero IV. Deve ter 1 ou mais.
2. Restrição legal: Todo IV deve pertencer a exatamente uma NF.
3. Restrição legal: Todo IV se referirá a exatamente um produto ou serviço[2].
4. Restrição legal: Um P/S deve sempre pertencer a um IV ou a um outro P/S.
5. Restrição legal: só podem ser adicionados à uma NF, P/Ss que estejam cadastrados no BD:P/S (Banco de Dados de Produtos e Serviços). Só o BD:P/S pode criar objetos P/S. O BD:P/S contém informação sobre produtos e serviços inclusive a categoria tributária de cada P/S especifico.
6. Restrição legal: Uma NF é criada no estado “em elaboração” e isto deve constar de uma eventual impressão.
7. Restrição legal: Uma vez que esteja completamente preenchida com todos os seus IV, uma NF deve ser validada (checa requisitos e calcula todos os impostos) e armazenada no subsistema BD:NF (Banco de Dados de NF), que também se encarrega de submeter na prefeitura[3]. Então, a NF deve passar para o estado “validada”, e deve ser então completamente imutavel. Nunca nenhum dado da NF inclusive de qualquer IV, deve ser modificado em uma NF validada. O BD:NF não deve aceitar nem validar nem gerar um ID para uma NF já validada, ou com dados inválidos. Se uma NF for corretamente validada, um objeto imutavel representando-a deve ser passado como resposta ao usuário-programador.
8. Requisito: Há um conjunto de varios impostos a serem aplicados em uma NF. Cada imposto possui uma aliquota default para produtos e serviços, e cada categoria tributária de P/S pode ter uma aliquota diferenciada[4]. O BD:P/S é mantido atualizado e confiamos nas aliquotas armazenadas.
9. Requisito: Deve ser fácil para o usuário-programador incluir um novo imposto. Deve haver uma interface padronizada para a programação de um novo imposto. Nos seus testes pode criar impostos simples, mas deve ser fácil programar a inclusao de qualquer novo imposto. Um novo imposto pode envolver cálculos arbitrariamente

[2] R2-3 implicam em um teste para demonstrar que não se pode cirar IV ou P/S sem “pai”. E assim por diante

[3] uma vez submetida, a NF tem valor legal e tributário, e não pode ser modificada, so cancelada. E mesmo que seja cancelada, consta nos livros e tem um ID único. Por isso a imutabilidade.

[4] Essa é a principal razão para usar P/S imutáveis. No BD garante-se que as categorias tributárias estão corretas. Ou se estiver errado, não é seu problema :-P Uma vez carregado um BD na memória, precisamos garantir que o programador não irá mudar a categoria tributária nem os detalhes do P/S..

complexos[5], mas sempre depende das quantidades, preços e categorias tributárias dos P/S. [DP Strategy, Command, Visitor]

10. Inclusive um imposto pode depender da sequencia de IVs e/ou P/S anteriores ou posteriores na mesma NF, portando deve ser possível ao imposto manter estado durante o processamento de uma sequencia de IVs. [DP Strategy, Command, Visitor]
11. Restrição Legal: Cada NF validada deve ter um identificador único, gerado durante a validação, que nunca pode se repetir[6]. Uma vez validada, esse ID deve aparecer em qualquer impressão.
12. ~~Restrição legal: Nota fiscal deve estar associada a exatamente um cliente pré-cadastrado no BD:CLI (Banco de Dados de Clientes).~~
13. Requisito do product owner[7]: deve ser facil de estender o sistema para especificar novas categorias de produtos e servicos no futuro, que ainda deverão ser associadas a um item de venda. [Acoplamento Abstrato]
14. Requisito do product owner: uma vez criada uma NF (antes da validação), os seus itens de venda devem ser modificados, adicionados ou deletados apenas pelos metodos apropriados. Deve-se cuidar que não haja acesso de escrita inapropriado a lista de itens por outros meios. [Acessor, ou seja, apenas cuidar do encapsulamento]
15. Requisito: Código de BD (mesmo que mockado), deve estar completamente separado, desacoplado do restante do sistema, e acessivel por uma API única, que é responsável por cadastros, buscas, submissões. Além disso, para simplificar logs, segurança, e desacoplamento, deve haver apenas um objeto (objeto, não classe) responsável por acessar o BD (isso representa uma restrição como "o seu aplicativo só pode fazer uma conexão com o BD"). [DP Façade ; Singleton]
16. Requisito: Código de calculo de impostos devem estar separados e desacoplados de forma a poderem ser modificados sem afetar o resto do sistema. [Strategy]
17. Requisito: Todas as entidades armazenados em BD devem corresponder a entidades imutáveis uma vez retirados do BD: ~~Clientes,~~ P/S, NFs validadas. [Imutable Object]
18. Restrição legal: Cada produto ou serviço (P/S) pode ser subdividido em outros produtos e/ou serviços. A quantidade de subdivisões depende do P/S específico. Por exemplo, S-Pintura sempre tem "S-Mao de Obra" e "P-Tinta". Não há limitação na profundidade das subdivisões[8]. Por exemplo, uma subclasse de "S-Mao de Obra" pode permitir um subcontratado, e o subcontratado sub-subcontrata outro, etc. A NF deve listar todas as subdivisões inclusive todas as folhas do último nível [Composite e Visitor].
19. Restrição Legal: O cálculo de um imposto pode depender não apenas dos IV e P/S de uma nota fiscal, mas também do conjunto e valores de NFs anteriores ao longo do tempo. Isso deve ser representado no código pela utilização de dados anteriores adicionais como entrada extra para o cálculo de um imposto. Cada imposto portanto pode definir um tipo de dados (classe) apropriado para os seus próprios cálculos,

[5] Talvez seria útil em algum país imáginário onde a legislação tributária é insana… :-)
[6] nesse trabalho pode ser um simples inteiro sequencial
[7] product owner é o "dono" do sistema. Por exemplo, quem te contratou para desenvolver.
[8] neste trabalho pode imprimir com uma pequena identação para indicar os níveis, e supor que cabe na página. Teste com alguns níveis de profundidade.

arbitrariamente complicado. Neste trabalho basta criar um classe associada a um imposto, e fornecer um objeto preenchido com valores anteriores ao calcular o imposto de uma NF. Um exemplo simples: o objeto-entrada contem a soma dos valores do imposto pagos no mes em todas as NF, e à medida que a soma aumenta, a alíquota para novas NF aumenta. Portanto o valor acumulado deve ser repassado em todos os calculos em cada NF. Note que não vale repassar só um float porque esse é um exemplo simples: a quantidade de dados e a complexidade dos cálculos poderiam ser muito maiores do que repassar e somar um valor. [Data Object]

Requisito Extra (entra no lugar do 12): incluir cliente, cpf, validador, não acrescentaria muito, então retirei. Mas suponha que a NF pode ter outras partes além de lista de IV e estado/ID, mesmo que nesse trabalho basta preencher um atributo "outros". Basta deixar o codigo extensivel para facilitar incluir outras partes arbitrarias na NF. Por exemplo suponha que o próximo passo depois desse trabalho seria incluir um subsistema para gerar clientes e incluir os atributos e validações necessários na NF. Mostre porque a sua solução facilitaria isso. [trivial se usou um padrão criacional apropriado para os outros requisitos]

Atributos dos Conceitos

Naturalmente há vários atributos para os conceitos:
Nota Fiscal: estado ("em elaboração" ou ID), valor (deve ser a soma dos valores dos itens), valores de cada imposto, condições e data de entrega, dados do cliente, informação da submissão.
~~Cliente: CPF, nome, endereco, telefone. Necessários para emitir a NF.~~
Item de venda: quantidade, desconto, condicoes e datas diferenciadas para o item. *Note que quantidade é um atributo do IV, não do P/S*.
Produto: nome, preco/unidade, setor responsavel, informacoes sobre o processo e materias-primas, categoria tributária, etc.
Servico: nome, preco/hora, setor responsavel, natureza (consultoria, treinamento, etc.), categoria tributaria.

Dicas e DPs para inspiração:

BDs: Façade e Singleton (Garantir ponto único de acesso). Os BDs em si são Mocks, mas deve ser um unico subsistema acessando esses Mocks.
Sugestao: use packages separados...

Mocks de GUI: Façade, Singleton, mediator se precisar comunicar entre diferentes pontos do codigo.

entidades armazenadas em BD: Immutable object[9]. Note que nunca deve-se criar nem ~~clientes~~, nem P/S, nem NFs validadas.

[9] Cuidado com a diferença entre atributos de tipos basicos (e.g. int) e objetos: um atributo "final int" nunca tem o seu valor modificado; Mas um atributo List, apesar de sempre apontar para a mesma lista, não garante que o conteúdo dessa lista permanece o mesmo.

P/S: Composite, com algum padrão criacional para um construtor. Algum tipo de recursão no construção para obter subitens do BD:P/S. Já fizeram criação de arvore recursiva, certo?

NF:
a criação é o ponto-chave, como tem varias partes um builder pode ser interessante;

Qualquer objeto criador (builder, construtor publico, ou factory method) não deve retornar uma nova NF se falta alguma coisa.

Uma NF é mutavel antes de validar e imutavel depois: uma forma de implementar e ser sempre imutável ;
Mas atenção à atributos Collection: se o atributo for final, sabemos que não podemos trocar a lista, mas podemos trocar o conteudo da lista. Se usarmos uma unmodifiable Collection (ortogonal a ser final ou não), garantimos que não podemos fazer add/delete, mas podemos modificar cada elemento.
Então:

- se fizer um clone de uma NF com collection normal e a nova usar copias unmodifiable, fechamos a NF
- se cada elemento for imutavel, então fica dificil mudar.
- poderiamos ser mais estritos e usar ImutableColections da lib Guava do Google, mas não precisa. Não quero criar mais dependencias.

Pode criar packages diferentes para subsistemas, de forma que alguns construtores sejam visiveis apenas dentro do package. Assim, se garante que clientes do package possam criar objetos apenas usando apropriadamente DPs como FactoryMethod, Builder, ou Static Factory Method. Ou seja, *pode considerar o usuario-programador como cliente do package e não só cliente da classe*. Se o programador mexer no seu package ele pode quebrar requisitos de criação. Isso significa que os demos podem estar em um package diferente, ou seja, o demo demonstra um usuario-programador usando o seu package.

Cada DP diagrama classes e nomes para os papeis de cada classe. Por exemplo, Observer e Subject aparecem no DP Observer. Se esses nomes são mantidos como parte dos nomes das suas classes, ou indicados com anotações em diagramas de classe ou sequência, é muito mais fácil de relacionar o seu código/texto com o DP sem precisar escrever ou explicar muito.