P1

Sistema de notas fiscais (Questão em 3 partes)

Obedecer as restricoes legais é critico. Uma auditoria examinará periodicamente todas as NF do sistema e falhas resultam em multa.

Conceitos (classes):

1. Nota Fiscal contem itens de venda.
2. NF contem informação do cliente.
3. Item de venda: Deve estar associado a um produto ou serviço pre-existente[1].
4. Cliente: dados de um cliente. contem CPF valido.
5. Cadastro: Acessa BD de clientes, possui metodos para encontrar clientes e para cadastrar novo cliente. Nao deve cadastrar cliente se o CPF for invalido.
6. VerificadorCPF: algoritmo que verifica se um CPF eh valido.

Restrições e Requisitos

1. Restrição legal: Nota fiscal não pode ter zero itens de venda. Deve ter 1 ou mais.
2. Restrição legal: Todo item de venda deve pertencer a exatamente uma nota fiscal.
3. Restrição legal: Todo item de venda se referirá a exatamente um produto ou serviço.
4. Restrição legal: Nota fiscal deve estar associada a exatamente um cliente pré-cadastrado.
5. Requisito do product owner[2]: 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.

---

[1] na verdade deveriam haver cadastros de produtos e servicos tambem, mas omitiremos por brevidade

[2] product owner é o “dono” do sistema. Por exemplo, quem te contratou para desenvolver.

6. Requisito do product owner: uma vez criada uma NF, 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.
7. Requisito do product owner: O objeto Cadastro é o único que pode criar Clientes, ou acessar o BD de clientes.

Atributos das Classes

Naturalmente há vários atributos para as classes:
Nota Fiscal: numero, valor (deve ser a soma dos valores dos itens), condições e data de entrega, alem dos dados do cliente.
Cliente: CPF, nome, endereco, telefone. Necessários para emitir a NF.
Item de venda: quantidade, desconto, condicoes e datas diferenciadas.
Produto: nome, preco/unidade, setor responsavel, informacoes sobre o processo e materias-primas, etc.
Servico: nome, preco/hora, setor responsavel, natureza (consultoria, treinamento, etc.)

[OBRIGATORIO] Parte I) Diagrama UML [2.0]

Defina o diagrama UML correspondente ao sistema, considerando as classes e requisitos/restrições acima (sem detalhar atributos). Para os seguintes relacionamentos, a notação UML deve indicar o tipo de relacionamento (associações, agregações, composições e heranças - vale criar classes auxiliares), e a multiplicidade das associações:
a) NF - Item
b) Item - Produto / Servico
c) NF - cadastro
d) Cadastro - ValidadorCPF

gabarito:

a) NF - IT : Composição NF --cria e contem---- 1..* IT
isso implica que NF deve ter uma lista de IT, e um item deve ser criado no construtor da NF. o construtor da NF precisa ter dados do produto/servico e do IT. (A nao ser que o IT tenha um mock do BD de servicos)
o construtor do IT precisa de um ponteiro para a NF

b) Item - Produto/Serviço
Precisa criar uma classe abstrata ProdServico, e entao:
Composição: IT ---cria e contem---- 1 P/S
No construtor do IT, precisa ter dados do produto servico. Ou, criar um Test Double criar um P/S.

c) NF - Cadastro: Agregacao (ou associacao) NF ---usa---- 1 Cadastro
O cadastro existe previamente, supostamente eh util em outras partes do sistema, e o NF precisa dele pra ser criado. O construtor do NF precisa de um cadastro, e usa o cadastro (que eh um test double tambem) para obter (ou pelo menos validar) um cliente.

d) Cadastro - ValidadorCPF
o cadastro precisa de um validador. O validador nao eh util fora do cadastro, ja que ninguem mais acessa o BD clientes. Entao eh Composicao, Cadastro cria um validador (que sera um test double) dentro do seu construtor.

Parte II) Código [2.5]

Implemente um esqueleto de codigo com as associações deste sistema considerando:

Não precisa detalhar campos das classes, nem no papel, nem na UML, nem no codigo. Os campos existem para explicar a necessidade das classes para o aluno e ajudar a dar contexto. Pode implementar uma classe vazia ou apenas um campo simplificado representando os diversos dados da entidade quando escrever os outros campos seria apenas adicionar algumas atribuicoes. Por exemplo, a classe Cliente pode ter apenas um inteiro CPF, e omitir nome e endereco, ou um formato complexo para o CPF por brevidade.
É apenas necessário implementar
i) um esqueleto demonstrando as especificidades dos tipos de relacionamento (associacoes, agregações, composição, multiplicidades), como nos exemplos na aula, ou seja, apenas o código implícito ou arquitetural, para a parte II
ii) o minimo necessario para passar nos testes, para a parte III, ainda sem precisar detalhar campos.

Parte III) Testes [2.0]

Cadastro de Clientes, verificacao de CPF: Numa implementacao real, precisaria acessar um BD e mostrar uma janela ou widget para listar, ordenar, buscar e selecionar um cliente, ou executar um algoritmo no caso do CPF, mas aqui deve usar um test double: use mock, stub, spy, ou dummy, o minimo necessario para passar os testes, e não implemente o BD ou GUI. Se quiser pode criar tambem um test double para um BD de produtos/serviços.
AO USAR UM TEST DOUBLE, INDIQUE E EXPLIQUE SE É UM MOCK, STUB, SPY, OU DUMMY
Implemente os seguintes testes:

a) cadastro de cliente (com CPF valido, precisa passar)
b) cadastro de cliente (com CPF invalido, precisa falhar)

c) cria NF valido e itens de venda (produtos e serviços);
    verifica se os itens estao corretamente presentes na nota fiscal
    verifica se o valor final da nota eh correto
    verifica se os dados do cliente e dos itens estao corretamente armazenados

d) cria NF com itens de compra, depois deleta itens e adiciona novos itens
    verifica se os itens estao corretamente presentes na nota fiscal
    verifica se o valor da nota eh correto
    verifica se os dados do cliente e dos itens estao corretos

e) crie uma NF com APENAS um item de compra

troque esse item por outro item
verifique se o item e o valor esta correto depois da troca.
Lembre que de forma nenhuma o sistema pode permitir que exista uma NF vazia.

Uso de ferramentas xUnit [1.5]

Usamos o JUnit, e existem outras ferramentas similares. Teça 3 comentários justificados sobre estas ferramentas, onde estes comentários podem ser:

a. positivos: explicar uma vantagem de utilizar estas ferramentas. Não vale simplesmente dizer “é bom para testar”. Deve ser uma qualificação destes testes, ou uma consequencia positiva de se usar a ferramenta.
b. negativos: explicar um defeito, limitação ou risco de se utilizar a ferramenta. Por exemplo, um contexto ou tipo de teste para o qual a ferramenta não é apropriada.

OS COMENTÁRIOS NÃO PODEM SER OS 3 DO MESMO TIPO.

Gabarito:

+ a automação dos testes permite executa-los a cada compilação, permitindo detectar bugs assim que estes são introduzidos.
+ as ferramentas possibilitam tecnicas como TDD e XP, já que não seria possivel realizar os testes necessários sem automação.
+ pensar nos testes (mesmo que não seja um TDD onde os testes são implementados antes do codigo de produção), tende a ajudar a não esquecer casos triviais ou especiais.
+ pensar nos testes permite que o programador verifique o projeto de classes. Esse eh realmente a melhor forma para usar essas classes? No caso do TDD, isso acontece antes da implementação.
+ as ferramentas diminuem a necessidade de atributos e métodos auxiliares para impressoes auxiliares, debug, e auto-testes das classes, ao separar e organizar os testes em outro arquivo. Assim o código de produçaõ fica mais limpo, direto e menor.
+ um teste pode servir como documentação auxiliar, provendo exemplos de uso das classes
- os testes introduzem mais uma carga de retrabalho em caso de uma grande refatoração. Afinal, se tudo mudou, e queremos manter os testes funcionando, temos que refatorar os testes também. (isso é apenas o lógico “no pain, no gain”)
- se iludir achando que se existem testes, nao existem bugs. Por exemplo, no kata dos numeros romanos, um de vocês encontrou um erro no teste 1999 que teria aparecido antes se houvesse sido testado o número 99. Implementar testes com cobertura total pode ser muito trabalhoso ou impossível na prática.
- xUnit: o nome indica testes de UNIDADE. Para testes de intregração, as ferramentas não são diretamente apropriadas. Veja em http://xunitpatterns.com/Unit%20Test%20Rulz.html

Michael Feathers of Object Mentor writes:
I've used these rules with a large number of teams. They encourage good design and rapid feedback and they seem to help teams avoid a lot of trouble.
A test is not a unit test if:

1. It talks to the database

2. It communicates across the network
3. It touches the file system
4. It can't run correctly at the same time as any of your other unit tests
5. You have to do special things to your environment (such as editing config files) to run it.

Tests that do these things aren't bad. Often they are worth writing, and they can be written in a unit test harness. However, it is important to be able to separate them from true unit tests so that we can keep a set of tests that we can run fast whenever we make our changes.
Michael Feathers

Ou seja, qualquer teste mais demorado ou que depende de outra infra-estrutura quebra a premissa de que os testes são executados transparentemente e imediatamente a cada compilação! Alem disso, quem quer depender de uma compilação que demora 10 segundos? Ou que depende de existir varios arquivos de entrada de teste no file system?

TDD e Repetiçoes: [1.5]

Joaozinho, ao implementar uma solução via TDD, começou implementando um caso trivial (um teste com entrada vazia, cuja implementação correspondente é apenas “return null”).
Ao seguir definindo mais testes e depois modificando a implementação com a transformação mais simples possivel para passar cada teste, o código adquiriu muitas repetições “copy-paste”, e longas estruturas switch-cases e cadeias if-elseif-elseif..., já que frequentemente a transformação mais simples para passar um teste era adicionar mais uma condição.
Comparando com a implementação da Mariazinha, que não seguiu TDD, vemos que a ideia geral de como resolver o problema é a mesma, mas no programa da Mariazinha a mesma logica foi implementada de forma mais curta e elegante, com loops, funções e fórmulas apropriadas que evitam as repetições e cadeias de condicionais do Joãozinho.
TDD realmente indica que, no passo de implementação do ciclo TDD, ao implementar uma nova funcionalidade, devemos realmente implementar a solução mais simples mesmo que não seja elegante ou eficiente. Então: O código do Joãozinho é realmente o que deve ser gerado pelo processo de TDD? **SIM** ( ) **NÃO** ( )
Se sim, explique qual o conceito e propósito do processo TDD nesse caso.
Se não, explique qual foi o erro de Joaozinho, em termos do processo TDD, utilizando a terminologia e considerando as 3 fases do ciclo do processo TDD: red->green->refactoring, ou seja, novo teste->nova implementação->refatoração.

os passos de teste e implementação de Joãozinho realmente foram corretos conforme o TDD indica. Mesmo que a implementação mais simples seja deselegante, ela é rápida de implementar e confirma que o problema foi compreendido e a solução funciona, e isso antes de gastar tempo com uma implementação mais elegante. Mas o próximo passo do ciclo TDD é justamente refatorar, ou seja, melhorar o codigo, por exemplo percebendo repetições e trocando-as por uma implementação mais geral e elegante. E como a implementação

"burra" já funcionou, já se sabe onde se quer chegar. Joãozinho simplesmente ignorou um dos 3 passos do ciclo TDD: o passo de refatoração!

Um Dalmatian é um Dog [1.5]

Dado que Dalmatian é subclasse de Dog, em Java não é permitido realizar a seguinte atribuição:

```
...
Dalmatian dalmatian = new Dog ( );
```

Onde a classe Dalmatian é subclasse da classe Dog.
Em termos de implementação, a razão é porque o tipo estático do objeto dalmatian é Dalmatian, e podem existir métodos em Dalmatian que não tenham sido implementados na classe do tipo dinâmico de dalmation, a saber, Dog. Na hipótese da atribuição da questão estar correta, seria perfeitamente legal enviar uma mensagem ao objeto dalmatian referente a método especificado apenas na classe Dalmatian – sem erro de compilação, pois o método é de classe estática Dalmatian –, mas não na classe Dog. Isso causaria um erro de tempo de execução!
Joãozinho esta especificando a nova linguagem JABBA (Joaozinho's Amazingly Beautiful oBject lAnguage). Em JABBA, se pretende que o problema acima seja resolvido da seguinte forma:
Ao se chamar um metodo especifico da sub-classe, se o tipo dinâmico do objeto for a superclasse, o objeto será "promovido" à subclasse através de um construtor default. Por exemplo, se a proxima linha for:
dalmatian.countBlackSpots(); // only the class Dalmatian has this method, not Dog
o objeto será transformado em Dalmatian, com o numero de spots default (todo atributo de classe em JABBA tem um valor default).
Supondo que seja possivel implementar a linguagem dessa forma, isso é uma boa idéia conceitualmente em termos de OO? **SIM** ( ) **NÃO** ( ) ***Explique.***

o problema conceitual é que a relação "é-um" de herança foi quebrada. Um dalmatian é um dog, mas um dog não é um dalmatian. Exemplo de potenciais problemas:

```
Dog dog1 = new Dog();
Dog dog2 = new Dog();
// muitas operações com estes dogs.
Dalmatian dalm1 = dog1;
Dalmatian dalm2 = dog2;
// … mais operacoes..
dog1.countBlackSpots(); // dog1 eh transformado em dalmatian, mas dog2 nao.
```

primeiro, fica confuso seguir as trasnformacoes implicitas. Como sabemos qual o tipo dinamico de cada variavel apenas lendo o codigo? Temos que seguir a sequencia de comandos?
Outro problema, supondo que o programe continue e que Rotweiller tambem extende Dog:

```
Rotweiller rot1 = dog1; // isso pode? como isso funciona se dog ja eh Dalmatian?
Rotweiller rot2 = dog2;
```

// e se fizer agora:
dog1.Kill(); // only rotweillers kill. but dog1 was a dalmatian
dog2.Kill(); // only rotweillers kill. dog2 still was a dog
como sabemos o tipo dinamico (dog? dalmatian? outro dog?), e como transformar dog1?
Se Rotweiller e Dalmatian possuirem atributos/metodos com o mesmo nome, o que fazer?
Mesmo que haja uma soluçao, isso implica em mais regras e restrições que complicam a semântica da linguagem de uma forma estranha para conhecedores de OO, alem de complicar a implementação do compilador.

Classe Abstrata [1.0]

Uma classe abstrata em Java pode não ter nenhum método abstrato? **SIM** ( ) **NÃO** ( )
Se sim, mostre com um exemplo simples porque isso seria útil. Senão, ilustre com um exemplo simples qual seria um problema (por exemplo, ser inútil, difícil de implementar, ou conceitualmente errado).

SIM. Porque cada metodo da classe abstrata pode ter uma implementacao (mesmo que vazia) e as classes filhas podem usar estas implementacoes.
Isso implica que eh possivel que a classe filha (concreta) seja exatamente a classe pai com outro nome, tudo bem.
suponha que Dog seja abstrata e implemente algumas coisas (latir, comer, andar)
Algumas raças latem diferente (overriding), outras tem metodos extra, mas algumas raças apenas herdam o Dog “padrao” com o comportamento “padrao”.
Claro que existe alguma diferença entre as raças alem do nome, mas nem todas essas diferenças foram representadas no codigo.

[OBRIGATÓRIA] Polimorfica [2.0]

No trecho de código Java abaixo, claramente a classe SubClass estende a classe SuperClass. /Total 2.0/
a) Quais são as classes estática e dinâmica das variáveis supC, subC e supC nas linhas 01, 03 e 05, respectivamente? /0.5/

Linha	Variavel	Classe Estática	Classe Dinâmica
01	supC		
03	subC		
05	supC		

Gabarito

Linha	Variavel	Classe Estática	Classe Dinâmica
01	supC	SuperClass	SuperClass
03	subC	SubClass	SubClass

05	supC	SuperClass	SubClass

[b(1)] Nas linhas 02, 04 e 06, que valor a variável local x recebe ao final de cada atribuição? /0.5/

Linha	Variavel	Valor
01	supC	
03	subC	
05	supC	

[b(2)] Explique, para cada uma dessas linhas, usando os conceitos de "classe estática", "classe dinâmica" e outros pertinentes, por que é assim e por que a atribuição compila corretamente? /0.5/

Linha	Variável Local	Valor	[b(1)]	[b(2)]
02	x	2	Como a variável supC tem classe estática e classe dinâmica SuperClass, o método amarrado dinamicamente é o add(int a, int b) da classe SuperClass	A variável supC tem classe estática e classe dinâmica SuperClass, que possui método add(int a, int b)!
04	x	10	Como a variável subC tem classe estática e classe dinâmica SubClass, o método amarrado dinamicamente é o add(int a, int b)da classe SubClass!	A variável subC tem classe estática e classe dinâmica SubClass, que possui método add(int a, int b)!
06	x	10	Como a variável supC tem classe estática SuperClass e classe dinâmica SubClass, o método amarrado dinamicamente é o add(int a, int b)da classe SubClass!	A variável supC tem classe estática SuperClass e classe dinâmica SubClass; o método a ser amarrado dinamicamente é o método add(int a, int b) da classe SubClass; compila corretamente porque a classe estática de supC, SuperClass, superclasse de SubClass, tem um método add(int a, int b) com mesma assinatura!

Obs.: Neste caso, era obrigatório responder 'usando os conceitos de "classe estática", "classe dinâmica" e outros pertinentes', como está na questão. Outros pertinentes incluem herança, overriding e polimorfismo. Mas tinha que usar na argumentação os conceitos de "classe estática" e "classe dinâmica"; sem eles a maior parte dos pontos

deve ser tirada: perda parcial, se falou bem apenas com herança, overriding e polimorfismo, e perda total, caso contrário!

1. Examinando os métodos das duas classes, pode-se dizer que o princípio da substituição se preserva? Justifique sua resposta! /0.5/

Resp.: De acordo com o princípio da substituição, um tipo A é dito ser um subtipo de um tipo B se uma instância do tipo A pode tomar o lugar do tipo B sem nenhum efeito observável. No caso, o princípio da substituição não se preserva, porque a subclasse SubClass, com base nos métodos add(~) e subtract(~), não "é uma classe SuperClass, ou seja, SubClass não é um subtipo de SuperClass. O método add(~) de SubClass corresponde ao método subtract(~) de SuperClass e o método subtract(~) de SubClass corresponde ao método add(~) de SuperClass. Na linha 06, o efeito observável é exatamente o contrário do add(~) da SuperClass, classe estática da variável supC, pois houve overriding por substituição neste caso, em que a propriedade intrínseca de soma é trocada pela propriedade intrínseca de subtração.

```
public class SuperClass{
        public int add(int a, int b) { return a - b; }
        public int subtract(int a, int b) { return a + b; }
...
}

public class SubClass extends SuperClass{
        public int add(int a, int b) { return a + b; }
        public int subtract(int a, int b) { return a - b; }
...
}
Cliente de SuperClass e SubClass
...
SuperClass supC = new SuperClass( );        // linha 01
int x = supC.add(6,4);                      // linha 02
...
SubClass subC = new SubClass( );            // linha 03
x = subC.add(6, 4);                         // linha 04
...
supC = subC;                                // linha 05
x = supC.add(6,4);                          // linha 06
...
```