UAST
Unidade Acadêmica de Serra Talhada - PE
UFRPE
Desde 2006
BSi
Bacharelado em Sistemas de Informação

MPOO

Site: https://sigs.ufrpe.br/sigaa/ava/index.jsf

Disciplina: Modelagem e Programação Orientada a Objetos (MPOO)

Profº: Richarlyson D'Emery

LISTA DE EXERCÍCIOS XIII

Leia atentamente as instruções gerais:

- No Eclipse crie um novo **projeto chamado br.edu.mpoo.listaXIII.SeuNomeSobrenome**, o qual deverá ter **pastas de pacotes** para cada questão: questao1, questao2, e assim sucessivamente, contendo todas as respostas da lista.
- Quando a questão envolver uma discussão teórica utilize um arquivo .txt (Menu File -> Submenu New -> Opção File), por exemplo, questao1.txt
- A lista envolve questões práticas e conceituais.

Você Sabia?

Nas Listas de Exercícios VI e VII, discutimos o conceito de Herança. Vimos que em Java só permite a herança simples, diferentemente de outras linguagens como, por exemplo, Python que permite que uma classe herde de mais de uma classe, ou seja, permite a implementação de herança múltipla. Também vimos o problema do diamante decorrente da herança múltipla; e que Python utiliza Method Resolution Order para a problemárica de herdar propriedades completamente diferentes da hierarquia de classes.

Em Java, uma estratégia para contorna a ausência de herança múltipla é a utilização de interface, mas essa herança múltipla será para comportamentos e constantes, pois a interface possui apenas essas duas definições.

```
public interface Interface {
     public static final int CONSTANTE=1; //uso das palavras static e final é opcional
     public abstract void metodoAbstrato(); //uso da palavra abstract é opcional
}
```

Usar **private** é **erro de sintaxe**!

Outra característica está na possibilidade de haver herança múltipla entre interfaces:

```
public interface Interface extends Interface1, Interface2, ..., InterfaceN {}
```

Implementar uma interface é como assinar um contrato com o compilador que afirma "Irei declarar todos os métodos especificados pela interface ou irei declarar minha classe abstract". Logo, é obrigatório adicionar os métodos da Interface em cada classe que a implemente, mesmo que um deles não precise executar nada, é necessário manter ao menos a sua assinatura na classe (*Respeito ao contrato!*).

A boa prática de programação, tem como convenção de nomenclatura para interface:

- Deve começar com a letra maiúscula;
- Deve ser um adjetivo como Runnable, Remote, ActionListener, etc.; e
- Use palavras apropriadas, em vez de siglas.

Fique atento!

- Quando uma classe realiza uma interface, a obrigatoriedade de `@Override` de métodos abstratos é dada se esta classe for concreta. Mas, se uma classe que realiza a interface for abstrata, então é opcional a sobreposição dos métodos abstratos dessa interface. Sendo assim: (i) se aplicada a sobreposição, os descendentes da classe abstrata que realiza a interface, não são obrigados a implementar os métodos abstratos; e (ii) se não aplicada, classes concretas descendentes passam a ter a obrigatoriedade de `@Override` dos métodos abstratos.

- *Use estratégias para evitar a repetição de código*! Chega um momento em que os métodos implementados da interface em várias classes tornam-se idênticos e repetitivos, levando o desenvolvedor a copiar e colar código para ganhar tempo de codificação, replicando métodos iguais de uma classe para outra. Até o Java 7, a melhor solução para não cair nessa prática era o uso de padrões de projetos (por exemplo: Strategy) que permitem reaproveitar métodos já implementados. Pensando nesse problema, em Java 8 foi introduzido o recurso Default Methods, deixando de ser necessário implementar todos os métodos da interface, o que possibilita redução no tempo gasto com o desenvolvimento e simplifica o uso desse tipo de estrutura:

```
public interface I{
    public default void metodo(){
          //corpo do método
    }
}
```

  Também pode-se definir o método como **static**, mas o método passa a pertencer a Interface:

```
public interface I{
    public static void metodo(){
          //corpo do método
    }
}
```

- Apesar de um objeto poder ser do tipo Interface (relacionamento "*é um*"), essa não é utilização recomendada, ou seja, tentar tratar o tipo de um objeto como Interface ao invés de Classe. Interfaces definem e padronizam como coisas, pessoas e sistemas podem interagir entre si. Objetos de software também se comunicam via interfaces. Uma interface Java descreve um conjunto de métodos que pode ser chamado em um objeto para instruí-lo, por exemplo, a realizar alguma tarefa ou retornar algumas informações.

Mão na Massa!

1) A partir dos diagramas de classes ao lado explique, em questao1.txt, qual a diferença entre eles, em especial destacando sobre a implementação do método definido na interface.

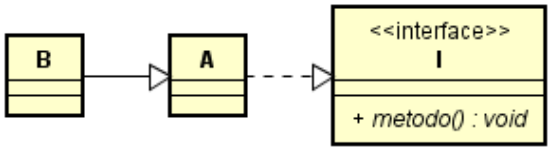


Diagrama A

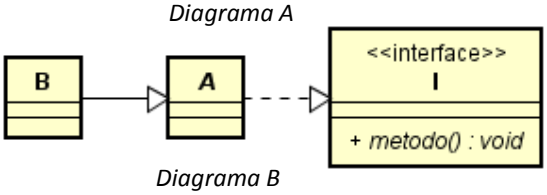
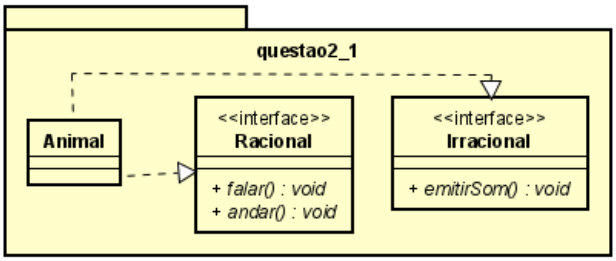


Diagrama B

2) A partir do diagrama de classes ao lado:



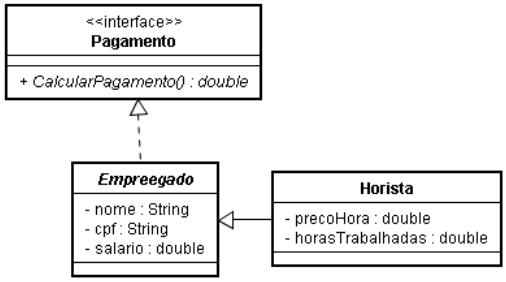
2.1) Codifique, em App.java, as seguintes situações:

- Um animal que fala e emite som.
- Um animal (homem) do tipo racional que apenas fala.
- Um animal (rato) do tipo irracional que apenas emite som.

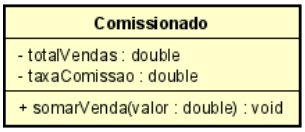
2.2) Nessa situação podemos dizer que um homem além de ser um animal racional também é um animal irracional, bem como um rato além de ser um animal irracional também é um animal racional? Em caso afirmativo, proponha uma solução fazendo uso de Classe Abstrata, Método Abstrato, Herança e Interface de maneira que um homem seja apenas um Animal Racional e um rato seja apenas um animal irracional. Apresente o diagrama de classes e a codificação Java.

Obs.: a codificação das questões abaixo deve estar nos pacotes questao2_1 e questao2_2, respectivamente.

3) Analise o diagrama de classes abaixo, as situações descritas (que deverão atualizar o diagrama) e codifique em Java:

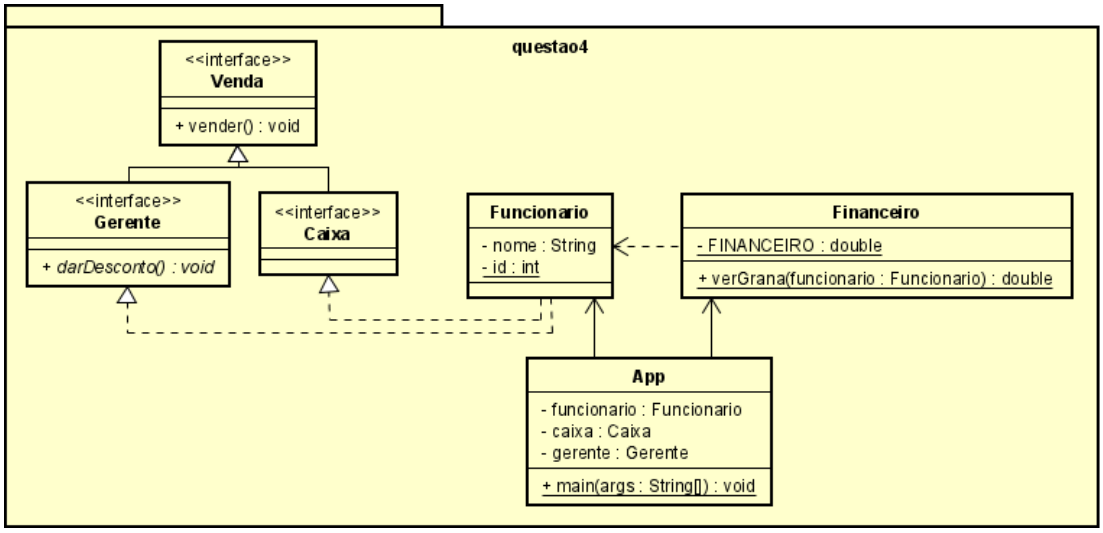


- O salário de uma instância de um horista que trabalhou 120h, sendo R$ 100,00 o valor da hora trabalhada.
- E se tivéssemos o Comissionado? Este tem seu salário calculado a partir de uma taxa de comissão sobre o total de vendas realizado.



- Como fazer para diferenciar a exibição dos dados desses empregados por um método? Apresente sua solução e exiba os dados dessa instância através de uma String em console.
- Apresente o novo diagrama de classes a partir das soluções codificadas.

4) Analise o diagrama de classes da questão em que ilustra a utilização de interface e herança:



4.1) Implemente o diagrama em Java. Crie uma aplicação (App) que contém instâncias para Funcionario, Caixa e Gerente.

4.2) Descreva em questao4.2.txt, o que acontece com o sistema quando o id de Funcionário é tido como estático? Sua resposta deverá está embasada e demonstrada por codificação (apresente-a em App da questão 4.1).

4.3) Quais os comportamentos que um caixa poderá invocar? Analise a corretude do diagrama e explique-a em questao4.3.txt.

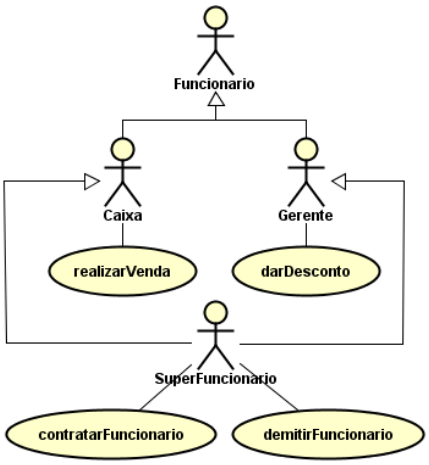
4.4) Dada a codificação para o método verGrana de Financeiro:

```
public static double verGrana(Funcionario funcionario){
     if(funcionario instanceof Gerente)
          return FINANCEIRO;
     return 0.0;
}
```

Observa-se que apenas um Gerente poderia ver o conteúdo Financeiro. Entretanto, isso não garante que apenas esse funcionário tenha acesso à informação. Além de um Funcionario e um Gerente, ilustre na App da questão 4.1 a codificação de como um caixa poderia se passar por um Gerente para ter acesso a tal informação?

4.5) Apresente uma solução (diagrama de classes e codificação Java) que contorna essa problemática, ou seja, que só um Gerente possa acessar essa informação. Obs.: O método verGrana não poderá ser modificado e ou definido em outra classe.

5) Uma Empresa solicitou a um de seus programadores (*de codinome mustela putórius furo – "O Furão"*) que resolvesse uma demanda de funcionalidades de seu sistema. Esta demanda envolve os atores Caixa, Gerente e o Proprietário da empresa (que chamaremos de SuperFuncionario). Cada ator detém comportamento(s) específico(s) conforme sua função ilustrada no diagrama de use case ao lado:

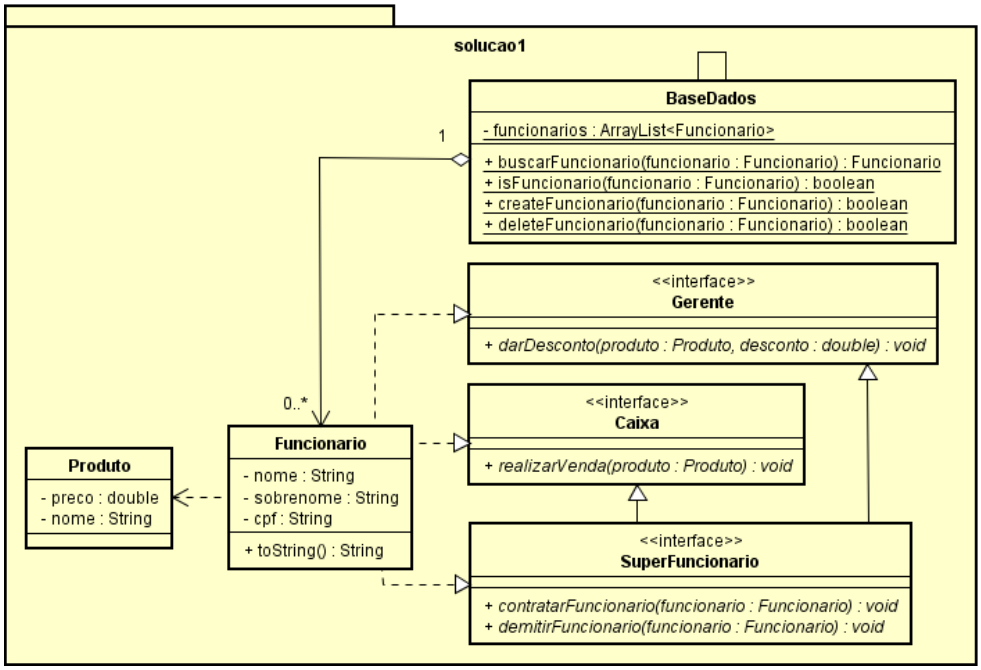


Sabendo que o sistema da empresa é desenvolvido em Java, então a solução deve utilizar Interface ou Herança (simples ou múltipla)? Antes de responder analise as situações que se segue. Em todas essas, deve-se respeitar as seguintes regras de negócios:

- RN01 – Um funcionário só poderá ser contratado se tiver fornecido nome completo e cpf;
- RN02 – Um funcionário já contratado não poderá ser contratado novamente;
- RN03 – Um funcionário só poderá ser demitido se já tiver sido contrato pela empresa;
- RN04 – Um funcionário só pode executar os comportamentos definidos;
- RN05 – Nenhum funcionário poderá ser tratado como outro, ou seja, o tipo caixa não poderá ser um gerente ou superfuncionario; o tipo gerente não poderá ser um caixa ou superfuncionario; e o superfuncionario não poderá ser um caixa ou gerente; e
- RN06 – A codificação deve aproveitar comportamentos já definidos, evitando a duplicidade de programação.

Obs.: Para as situações abaixo utilize a estrutura de pacotes questao5.situacao1 para a situação 1, questao5.situacao2, e assim sucessivamente. Logo aproveite a codificação copiando-a e colocando-a, por exemplo: copiar o pacote "questao5.situacao1" e colar no projeto renomeando para "questao5.situacao2".

5.1) **(Situação1)** Assumindo que o programador tenha optado por uma solução de interface, conforme o diagrama de classes abaixo, então realize sua codificação e analise se o programador solucionou corretamente apontando problemas relacionados às regras de negócio definidas. Faça sua análise apontando em comentários suas considerações em uma aplicação (App.java) a partir da criação dos funcionários na base e da chamada de métodos.



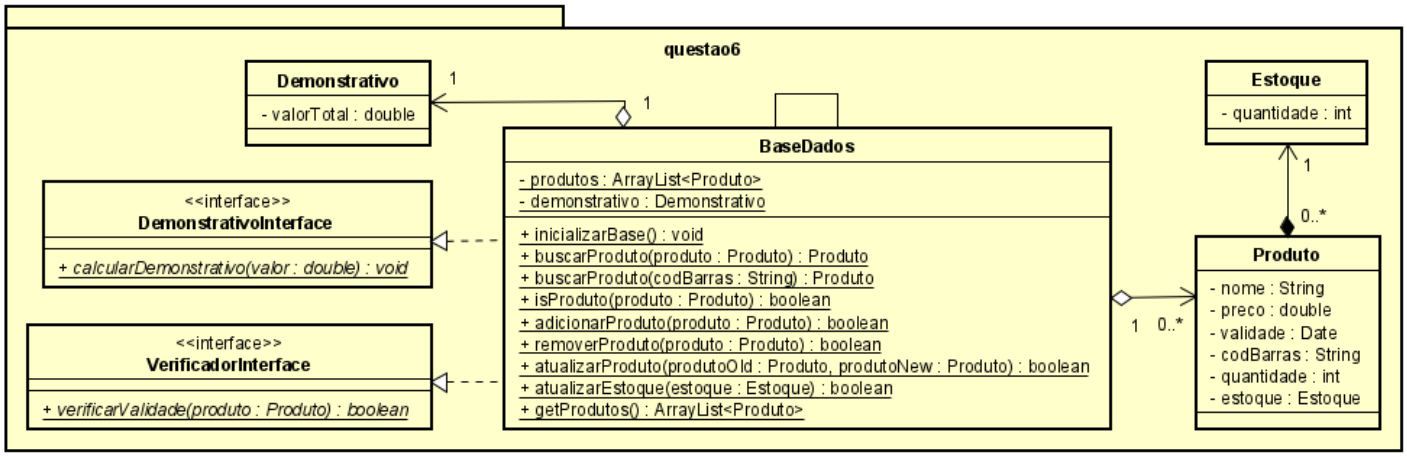
5.2) **(Situação2)** Tendo o programador passado inúmeras horas para concretizar a solução 1 e no desespero de não passar vergonha e ser demitido, elaborou a solução 2, substituindo apenas a definição da base de dados conforme a codificação "BaseDados.java". Você acha que a nova solução está correta? Aproveitando a codificação da Situação 1, modifique-a de maneira a contemplar a nova solução (vide dica de copiar codificação para novo pacote). Faça sua análise apontando em comentários suas considerações em uma aplicação (App.java) a partir da criação dos funcionários na base e da chamada de métodos.

```java
//BaseDados.java
package questao4.situação2;

import java.util.ArrayList;

public class BaseDados {

    private static ArrayList<Caixa> caixas = new ArrayList<Caixa>();
    private static ArrayList<Gerente> gerentes = new ArrayList<Gerente>();
    private static ArrayList<SuperFuncionario> superFuncionarios = new ArrayList<SuperFuncionario>();

    /* funcionarios: ArrayList<Funcionario>
     * Servirá apenas para facilitar a manutenção das operações da base!
     */
    private static ArrayList<Funcionario> funcionarios = new ArrayList<Funcionario>();

    public static Funcionario buscarFuncionario(Funcionario funcionario){
            //corpo do método
    }

    public static boolean isFuncionario(Funcionario funcionario){
            return funcionarios.contains(buscarFuncionario(funcionario));
    }

    private static boolean isDados(Funcionario funcionario){
            //corpo do método
    }

    public static boolean createFuncionario(Caixa caixa){
            if (isDados((Funcionario)caixa))
                    if (!isFuncionario((Funcionario)caixa)){
                            funcionarios.add((Funcionario)caixa);
                            return caixas.add(caixa);
                    }
            return false;
    }
// (codificação continua na próxima página)


    public static boolean createFuncionario(Gerente gerente){
            //corpo do método semelhante a createFuncionario(caixa:Caixa):boolean
    }

    public static boolean createFuncionario(SuperFuncionario superFuncionario){
           //corpo do método semelhante a createFuncionario(caixa:Caixa):boolean
    }

    //delete não implementado

    public static ArrayList<Caixa> getCaixas() {
            return caixas;
    }

    public static ArrayList<Gerente> getGerentes() {
            return gerentes;
    }

    public static ArrayList<SuperFuncionario> getSuperFuncionarios() {
            return superFuncionarios;
    }
}
```

5.3) **(Situação3)** Aproveite a oportunidade e mostre que você tem a capacidade de ocupar a vaga de programador sênior na empresa apresentando uma solução (*diagrama de classes* e *codificação Java*) adequada para o problema. Lembre-se de respeitar a descrição do problema (use case e regras de negócios) para as tentativas de soluções do programador "O Furão". *Sua solução deve utilizar obrigatoriamente o conceito de Herança e Interface*.

Se sua solução estiver de acordo com o desejado pelo Prof. Richarlyson (seu *Product Owner*), você poderá ganhar **até 1 ponto na média na 2ª V.A.**! (Lembre-se de informar se sua solução está de acordo após a apresentação da solução do *Product Owner)*

6) Analise o diagrama de classes abaixo e as regras de negócio:



São regras de negócios:

- RN01 – Todo produto é armazenado na base de dados;
- RN02 – A manutenção da base é tida pelos métodos CRUD;
- RN03 – A quantidade de itens de um mesmo produto é tida pela quantidade disposta em estoque;
- RN04 – Quando um item de um produto está vencido este deve ser descartado, ou seja, retirado da base através do método atualizarEstoque da base; e
- RN05 – Quando um item é removido do estoque, seu valor deve ser atribuído como prejuízo ao valorTotal do demonstrativo.

6.1) Codifique em Java o diagrama e em uma App:

- Que contém 100 itens do produto Feijão de 1kg que custa que custa R$ 6,00, código de barras "COD0001". Sendo 50 desses com validade de 31/12/2021 e 50 unidades com validade 31/12/2020.
- Que contém 200 itens do produto Fubá de 500g que custa que custa R$ 1,50, código de barras "COD0002". Sendo 100 desses com validade de 31/12/2021 e 100 unidades com validade 31/12/2020.
- Que contém 100 itens do produto Macarrão de 500g que custa que custa R$ 2,00, código de barras "COD0003". Sendo 80 desses com validade de 31/12/2021 e 20 unidades com validade 31/12/2020.
- Assumindo o fato que nenhum item foi vendido, mas diversos itens estão com a validade vencida, exiba em console o valor do prejuízo acumulado dessa empresa.

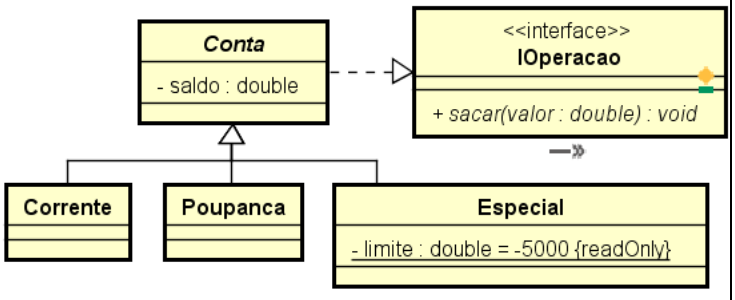
6.2) Comente em questao6.2.txt, o que precisaria ser feito para que os métodos das interfaces deixassem de ser `static`?

7) Em "Fique Atento!" foi destacada a possibilidade de utilização do padrão de projetos Strategy como solução para evitar a duplicidade de código desnecessário pela utilização de Interface. *Isso mostra a importância de cursar a disciplina optativa Arquitetura de Softwares no curso de BSI!* Sendo assim pesquise sobre o padrão Strategy e apresente um resumo escrito (questao7.txt) contendo definições e aplicações (ou um vídeo-exemplo prático com codificação Java, máximo de 15 minutos). Independente de sua escolha, não se esqueça de comentar as vantagens e desvantagens para o uso de Strategy.

Desafio

8) **(Opcional):** Em "Fique Atento!" foi destacada a possibilidade de utilização do padrão de projetos Strategy como solução para evitar a duplicidade de código desnecessário pela utilização de Interface. Isso mostra a importância de cursar a disciplina optativa Arquitetura de Softwares no curso de BSI! Sendo assim pesquise sobre o padrão Strategy. O diagrama de classes abaixo aponta um cenário em que Strategy poderia ser utilizado.

Descrição do Problema: Sabendo a existência de diferentes tipos de contas, observamos o fato de que nestas podemos realizar operações de saques, ou seja, a possibilidade de retirar valores do saldo destas contas. Quando uma conta é do tipo Especial, as retiradas podem ser realizadas de maneira a tornar o saldo negativo, mas limitada a um limite, enquanto nas demais só pode realizar um saque enquanto houver saldo positivo. A partir do problema:



8.1) Codifique em Java o diagrama do problema sem o uso de Strategy. Em App.java crie uma instância para cada tipo de conta com saldo inicial de R$1000,00 e realize 100 saques de R$ 100,00. Comente em questao8.1.txt onde o padrão de projetos poderia ser aplicado.

8.2) Proponha uma solução (diagrama de classes e codificação) através de Strategy.

8.3) A partir da sua experiência comente em questao8.3.txt as vantagens e desvantagens para o uso de Strategy.