PC-2 / LP-2 2009/2 – Lista 2

Prof. Alexandre Sztajnberg

Em todos os exercícios a(o) aluna(o) deve, além de atender aos requisitos enunciados, utilizar os conceitos e características de orientação a objetos do Java:

- Encapsulamento (incluindo modificadores de acesso),
- Herança (de classe e interface) e polimorfismo;
- Sobrecarga / Sobreposição de métodos;
- Geração / Captura e Tratamento de exceções;
- Uso das classes básicas (*Object*, por exemplo);
- Classes / pacotes
- Leitura de dados via fluxo de entrada

(ou seja, boas praticas de programação orientada a objetos devem ser empregadas mesmo se não foram explicitamente solicitadas)

Exercício 1: Banco de Dados - Locadora

Uma Locadora de Vídeo precisa ter controle sobre o cadastro de clientes e acervo de filmes disponíveis para locação. Desenvolva uma aplicação que permita criar um cadastro de clientes e de filmes, e acessá-los quando necessário.

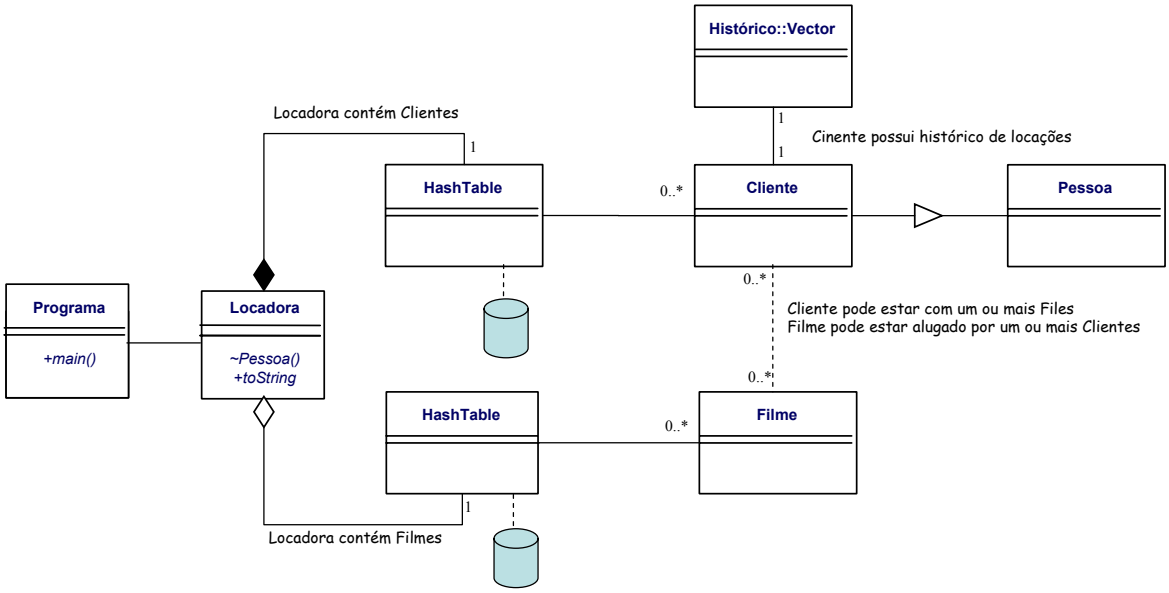

Figura 1: modelo da Locadora

Requisitos:

I) Crie uma classe *Pessoa* com os campos *nome* e *dataNasc* (da classe *String*), que devem ser inicializados pelo construtor.

II) Crie uma classe *Cliente*, que estende *Pessoa*, e possua um campo *String endereco* e um campo do tipo *int codigoCliente*. O construtor da classe *Cliente* deve inicializar todos os seus campos.
A classe *Cliente* também deve ter um campo *Histórico* da classe *Vector*. Este campo deve armazenar objetos da classe *Aluguel.*

A classe *Aluguel* deve conter a data de locação, data de devolução e o código do *Filme* alugado.

A classe *Cliente* deve ter um método *adFilmeHist()*, que recebe a data de locação, data de devolução e o código do *Filme* alugado, cria um objeto *Aluguel* com estas informações e adiciona o mesmo no *Histórico*.

III) Crie uma classe *Filme*, com campos para:
- Código do Filme (caracteres alfanuméricos)
- Título do Filme
- Categoria (que pode ser Aventura, Ficção, etc.)
- Quantidade (inteiro)
- Alugados (inteiro)

... e o seguintes construtores:
- Um construtor que inicializa todos os seus campos
- Um construtor que recebe como parâmetro um objeto *String* e inicializa o campo do Título do Filme.

Também deve ter os seguintes métodos:
- *aloca*, que não recebe parâmetros e acerta o campo *Alugados*. Caso todas as cópias estejam alugadas, deve levantar a exceção confirmada *CopiaNaoDisponivelEx*;
- *devolve*, que não recebe parâmetros e acerta o campo *Alugados*. Caso nenhuma cópia tenha sido alugada, deve levantar a exceção *NenhumaCopiaAlugadaEx*.

IV) Implemente a classe *Locadora* com:
- um campo para o cadastro de clientes contendo um objeto da classe java.util.*HashTable*
- um campo para o cadastro de filmes contendo um objeto da classe java.util.*HashTable.*

A classe *Locadora* classe deve possuir dois construtores:
- um que inicialize os campos
- um construtor que carregue o cadastro de clientes e o cadastro de filmes salvos em dois arquivos distintos.

Locadora deve possuir os seguintes métodos:
- *cadastraCliente*: de retorno void, que recebe como parâmetro um objeto da classe *Cliente* e o armazena no objeto *HashTable correspondente*. O código do cliente deve ser utilizado como chave;

- *cadastraFilme*: de retorno *void*, recebe como parâmetro um objeto da classe *Filme* e o armazena no objeto *HashTable correspondente*; O código do Filme deve ser usado como chave;
- *salvaArquivo*: de retorno *void*, recebe como parâmetros um objeto da classe *HashTable (que pode ser o cadastro de clientes ou o acervo de filmes)* e um objeto da classe *String* contendo o nome do arquivo onde o outro parâmetro será salvo;
- *leArquivo*: de retorno *void*, recebe como parâmetros um objeto da classe *String* contendo o nome do arquivos a ser lido (como o construtor, mas que pode ser chamado a qualquer hora, e lê somente o acervo ou o cadastro de clientes);
- *alugaFilme*: recebe como parâmetros a referência a um objeto Cliente e a referencia a um objeto da classe *Filme*. (as referências já devem ter sido validadas – obtidas através dos métodos *getFilme* e *getCliente* – abaixo). Chama o método aloca no objeto *Filme* e atualiza o histórico no objeto *Cliente* chamando *adFilmeHist.*
- *String imprimeFilmes()*: Devolve uma string com a lista de filmes cadastrados, ordenados pelo título.
- *String imprimeClientes()*:Devolve uma string com a lista de clientes cadastrados, ordenados pelo nome.
- *Filme getFilme (int cód)*: Recebe o código do filme e obtém o objeto *Filme* da HashTable correspondente. Se o filme estiver cadastrado, deve gerar a exceção *FilmeNaoCadastradoEx*;
- *Cliente getCliente (int cód)*: Recebe o código do cliente e obtém o objeto *Cliente* da HashTable correspondente. Se o cliente não existir na HashTable, deve gerar a exceção *ClienteNaoCadastradoEx*;

Obs.: Além dos métodos listados, verifique se as classes *Pessoa*, *Cliente*, *Filme* e *Locadora*, devem ter, para cada campo, um método *get<NomeDoCampo>()* que retorna o conteúdo do campo.

Obs: Para todas estas classes, exceto *Locadora*, sobrescrever o método *toString()* para mostrar todas as informações sobre o objeto [no caso de Cliente, deve exibir inclusive o histórico].

As classes dos itens I, II, III e IV devem pertencer a um pacote *chamado pcii_gXX.locadora* (onde XX é o número do grupo).

V) Desenvolver o programa principal, que deve ter os seguintes módulos:

- *Manutenção*, que cria, abre e salva os arquivos (dois) que contém, cada um, uma HasTable (filmes e clientes);

- *Cadastro*, que cadastra clientes e filmes. Deve exibir um menu com opções para: cadastrar clientes, cadastrar filmes e salvar em arquivo. A opção de salvamento em arquivo deve exibir um sub-menu para que o usuário escolha se deseja salvar o cadastro de clientes ou o cadastro de filmes.

- *Locação*: que executa a locação de um filme para um cliente. Um menu com as opções de exibir o cadastro de filmes ou fazer uma locação deve ser exibido. Ao

fazer uma locação, deve ser obtido o objeto Filme e Cliente. As exceções (filme não existe, cliente não cadastrado, cópia não disponível, etc.) devem ser tratadas. Quando uma locação for bem sucedida os dados devem ser escritos na tela, o registro do filme mantido (e salvo na *HashTable* – dependendo de como for programado) e o histórico do cliente atualizado.

Obs: no programa principal, o programador deve customizar a política da locadora (por isso as classes de suporte estão em um pacote e o programa principal não está). Por exemplo, pode restringir o número máximo de filmes que um usuário pode alugar de uma só vez e o número de dias que um usuário pode ficar com a fita sem pagar novo aluguel.

- *Relatório*, que permite listar o acervo de filmes, o cadastro de clientes ou detalhes de um cliente específico.

VI) Desenvolver dois *scripts* (um para Windows e outro para Linux) para inicializar o programa. Os *scripts* podem (devem) usar argumentos e/ou variáveis de ambiente para carregar a JVM e classes adequadamente (isso deve tornar os pacotes "independentes de localização).

Exercício 2: *Applet* – Calculadora IMC

Disponibilize o programa de cálculo de IMC em um Applet.

Utilize campos de entrada de dados, *radio buttons* e *slide bars* para permitir o ajuste do peso.

Faça a previsão de mensagens de erro (em janelas pop-up ou campos de texto).

Utilize cores para apresentar o resultado (verde / vermelho)

O *applet* deve ser acessado pela Internet, através da URL: snarf.ime.uerj.br/~pcii_gXY

Para isso você vai precisar criar um diretório ~/public_html e um arquivo index.html com as permissões e o conteúdo adequado, além das classes em Java.

Exercício 3: *Threads*- Problema do Banheiro Unissex

Suponha que em um local movimentado haja apenas um banheiro, "Unissex". O banheiro possui em seu interior cabines individuais, mas homens e mulheres não podem usá-lo ao mesmo tempo.

Vamos utilizar técnicas de programação concorrente para controlar o acesso ao banheiro e simular essa situação.

Vamos, também, criar algumas possibilidades para esta solução, usando herança.

I) Crie as classes de exceção *BanheiroOcupado* e *BanheiroVazio* que herdem de *ArrayIndexOutOfBoundsException* e exibam, respectivamente, as mensagens "Desculpe - O banheiro esta lotado" e "O banheiro esta vazio". Crie as classes *Alerta* e *Espera* que herdem de *Exception* e exibam, respectivamente, as mensagem "Homens e mulheres não podem usar o banheiro ao mesmo tempo" e “Aguarde um momento por favor”.

II) Crie a classe *Humano* com os campos *gênero* (que pode ser masculino ou feminino), *identificador* (que é um valor inteiro) e um campo contendo uma referência para um objeto da classe *Banheiro* (que será definida abaixo). O construtor da classe *Humano* recebe como parâmetros um objeto da classe *Banheiro* e um valor inteiro, e inicializa os campos correspondentes.

III) Crie a classe *Banheiro* com campos inteiros *quantidade* e *capacidade* e o array *cabines*, onde se possa alocar instâncias da classe *Humano*. Seu construtor utiliza o campo capacidade para inicializar o tamanho do array, que deve ser 5. A classe *Banheiro* deve ter os seguintes métodos:

void entraBanheiro(Humano h): Homens e mulheres não podem usar o banheiro ao mesmo tempo, logo este método deve verificar o gênero (feminino ou masculino) de quem tenta entrar no banheiro, e verificar se dentro do banheiro há pessoas do sexo oposto. Caso haja, deve lançar a exceção *Alerta*, senão, o método tenta inserir o objeto no banheiro e em caso de sucesso imprime a mensagem "<Homem ou Mulher> <identificador> Entrou". Se o banheiro estiver lotado, deve lançar a exceção *BanheiroOcupado*.

void saiBanheiro(): Tenta retirar um dos objetos (Homem ou Mulher) que estão usando o banheiro. Caso o banheiro esteja vazio, lança a exceção *BanheiroVazio*.

Lembre-se de utilizar mecanismos que garantam que homens e mulheres tenham as mesmas chances de entrar no banheiro. A exceção Espera deve ser lançada nos casos em que as condições deste mecanismo não tenham sido atendidas.

IV) Crie a classe *BanheiroExt*, que estende *Banheiro* e modifica o número de cabines disponíveis (tamanho do banheiro). Seu construtor recebe como parâmetro o novo tamanho máximo do banheiro. Os métodos *entraBanheiro* e *saiBanheiro* devem manter as mesmas funções, mas serão sobrescritos da seguinte forma:

- Terão o modificador *synchronized* acrescentado à sua assinatura, para que seja resolvido o problema da concorrência.
- Para sincronizar o acesso ao banheiro, deve ser usada a seguinte "receita":

No início do método *entraBanheiro()*:

```
while(banheiro cheio)
{
     wait(); //Espera alguém sair
}
```

No início do método *saiBanheiro()*:

```
while(banheiro vazio)
{
      wait(); //Espera alguém entrar
}
```

No fim de ambos os método:

```
notifyAll();
```

V) - Crie as Classes *Homem* e *Mulher* que estendam a classe *Humano*, implementem a interface *Runnable* e tenham o campo de classe *status* (que pode ser utilizado para garantir justiça no acesso ao banheiro). O construtor de cada classe recebe como parâmetros um objeto da classe *Banheiro* e um valor inteiro, inicializa esses campos e também o campo gênero com as palavras "Masculino" para *Homem* e "Feminino" para *Mulher*. Em cada classe implemente os métodos:

private void entraBanheiro(): Tenta colocar um objeto (um homem para a classe *Homem* e uma mulher para a classe *Mulher*) no banheiro. A cada tentativa deve exibir a mensagem "Tentando entrar <Homem ou Mulher> <identificador>". A cada vez que entrar no banheiro, os homens e mulheres devem permanecer lá por um tempo aleatório (use: Thread.sleep((long)(Math.random()*100))). Se a entrada for bem-sucedida imprime a mensagem "<Homem ou Mulher> <identificador> Entrou". Caso ocorra alguma exceção do tipo *BanheiroOcupado* ou *Alerta*, deve "dormir" um tempo aleatório e tentar novamente.

private void saiBanheiro(): Tenta retirar um objeto do banheiro. A cada tentativa imprime a mensagem "Saindo <homem ou mulher> <identificador>". "<Homem ou Mulher> <identificador> Saiu". Caso capture uma exceção do tipo *BanheiroVazio*, "dorme" um tempo aleatório e tenta novamente.

public static boolean getStatus(): Retorna o status das instâncias da classe.

public static boolean setStatus(): Altera o status das instâncias da classe.

O método *run* deve consistir de um loop infinito, que em cada passo tenta colocar um homem ou mulher no banheiro, dorme alguns segundos, e em seguida tenta retirar alguém do banheiro.

VI) - O programa principal deve perguntar qual tipo de banheiro será criado: *Banheiro* ou *BanheiroExt* e também o número de homens e mulheres que tentarão utilizar o banheiro. O programa inicializa o "banheiro" com o tamanho adequado (fornecido pelo usuário no caso de *BanheiroExt* e default para Banheiro) e as instâncias de *Homem* e *Mulher*, que em seguida, terão sua atividades iniciadas, como *threads*.

Exemplo de Execução:

```
1 - Banheiro  2 - Banheiro Unissex
1

Numero de Mulheres: 8
Numero de Homens: 5

Tentando Entrar - Homem 2
Homem 2 Entrou
Tentando Entrar - Homem 4
Tentando Entrar - Homem 3
Tentando Entrar - Homem 1
Homem 4 Entrou
Tentando Entrar - Mulher 1
Tentando Entrar - Homem 5
Homem 1 Entrou
Homem 3 Entrou
Tentando Entrar - Mulher 5
Mulher 1: Homens e mulheres nao podem usar o banheiro ao mesmo tempo
Homem 5 Entrou
Mulher 5: Homens e mulheres nao podem usar o banheiro ao mesmo tempo
Homem 3 Saiu
Homem 4 Saiu
Homem 2 Saiu
Tentando Entrar - Homem 4
Tentando Entrar - Mulher 8
Homem 1 Saiu
Homem 4: Aguarde um Momento
Mulher 8: Homens e mulheres nao podem usar o banheiro ao mesmo tempo
Homem 5 Saiu
Tentando Entrar - Mulher 6
Tentando Entrar - Mulher 3
Tentando Entrar - Homem 4
Tentando Entrar - Mulher 8
Tentando Entrar - Mulher 5
Mulher 6 Entrou
Mulher 3 Entrou
Tentando Entrar - Mulher 2
Homem 4: Homens e mulheres nao podem usar o banheiro
Mulher 8 Entrou
Mulher 5 Entrou
Tentando Entrar - Mulher 7
Tentando Entrar - Mulher 1
Tentando Entrar - Mulher 4
Tentando Entrar – Homem 1
Mulher 2 Entrou
Mulher 7: Desculpe - O banheiro esta lotado
Mulher 1: Desculpe - O banheiro esta lotado
Mulher 4: Desculpe - O banheiro esta lotado
Homem 1: Homens e mulheres nao podem usar o banheiro ao mesmo tempo
```