*Introdução à Programação 2 (BC3 / BC4) - **Lista 2***
2020.1 (remoto)
Profs. Leandro M. Nascimento / Vanilson A. A. Buregio

1. (2,0 - Array/ArrayList) Quais são as suas chances de ganhar na Mega-Sena?

Você como exímio programador da UFRPE resolveu entender melhor as suas probabilidades de ganhar na mega-sena. Para isso, você deve criar um programa que gere um jogo de 6 números aleatórios, que vamos chamar de bilhete premiado e, em seguida, gere uma quantidade N de jogos com X números aleatórios (6<=X<=15). A ideia é que você verifique qual a quantidade N de jogos a serem apostados de tal forma a se acertar os mesmos números do bilhete premiado. Você também deve observar que, quanto maior o valor de X, referente à quantidade de números apostados por jogo, melhores serão as suas chances de ganhar, ou seja, menor será o número N de jogos. Ao término da execução do seu programa, as seguintes informações deverão ser apresentadas no seguinte formato:

- `Jogos de 6 números: você apostou 5464812 vezes para ganhar na mega-sena. Dentre esses jogos, você acertou 239 quadras e 69 quinas.`
- `Jogos de 7 números: você apostou 896138 vezes para ganhar na mega-sena. Dentre esses jogos, você acertou 198 quadras e 59 quinas.`
- `Jogos de 8 números: você apostou 224896 vezes para ganhar na mega-sena. Dentre esses jogos, você acertou 124 quadras e 55 quinas.`
- `Jogos de 9 números: você apostou 93449 vezes para ganhar na mega-sena. Dentre esses jogos, você acertou 432 quadras e 120 quinas.`
- `…(Repetir a mesma operação para os valores de 10 a 14)`
- `Jogos de 15 números: você apostou 112 vezes para ganhar na mega-sena. Dentre esses jogos, você acertou 96 quadras e 27 quinas.`

Essa sequência de informações lhe trará a noção clara de como suas chances de ganhar aumentam se a quantidade de números apostados por jogo aumenta.

Instruções: Crie uma classe para representar um Bilhete de Loteria. Essa classe deve conter como atributo um array de inteiros e um construtor que recebe como parâmetro o tamanho do array de inteiros e o inicializa com números randômicos de 01 a 60 (intervalo fechado). O array não pode conter números repetidos e deve manter os números de forma ordenada.

Implemente o método toString nesta mesma classe que retorna os números contidos naquele bilhete em uma String formatada, no seguinte formato:

`[07, 09, 14, 24, 39, 59]`

Implemente também nesta classe um método 'qtdNumerosContidos' que recebe um outro Bilhete de Loteria como parâmetro e verifica quantos números do bilhete entregue como parâmetro para o método estão contidos dentro do bilhete corrente (this), retornando esse valor como resposta.

Em uma classe à parte chamada de `MinhaLoteria`, implemente o seguinte procedimento:

- Instancie um objeto tipo `BilheteLoteria` que contém somente 6 números e armazene numa variável chamada de `bilhetePremiado`. Este objeto deve permanecer inalterado até o final do seu programa
- Crie um laço que vai variar a quantidade de números por cada bilhete apostado (6<=X<=15). Dentro deste laço, faça o seguinte:
  - Instancie uma variável do tipo `ArrayList<BilheteLoteria>` e chame-a de `bilhetes`.
  - Em um novo laço interno, instancie um objeto do tipo `BilheteLoteria`, chame-o de `bilheteSorteado` e armazene-o no `ArrayList`. O tamanho deste bilhete deve variar de acordo com qual iteração do laço externo o código se encontra (6<=X<=15).
  - Ainda dentro do laço interno, verifique quantos números do `bilhetePremiado` estão contidos no `bilheteSorteado`. Neste momento, você deverá usar variáveis auxiliares para armazenar o número de vezes que você encontrou uma quadra, uma quina ou uma mega-sena.
  - O laço interno somente deve parar quando uma mega-sena for encontrada e então as informações das quantidades de jogos, quadras e quinas são apresentadas (vide formato no início da questão).
- Com a intenção de facilitar a depuração do seu código, você pode imprimir a cada iteração do laço:

"`Sorteio de número YYY realizado. O bilhete sorteado [1, 2, 3, 4, 5, 6, 7] NÃO CONTÉM todos os números do bilhete premiado [2, 3, 4, 5, 6, 12]`"

  - O valor de YYY é um contador incremental de sorteios e os números devem ser os números armazenados nos bilhetes. Continue a executar o programa até que algum bilhete sorteado contenha todos os números do bilhete premiado.
  - Se o `bilheteSorteado` **contém** todos os números do `bilhetePremiado`, imprima "PARABÉNS VOCÊ GANHOU NA MEGASENA DEPOIS DE 'N' TENTATIVAS DE JOGO", substituindo o valor de N pelo número de tentativas e logo em seguida quais foram os números do `bilheteSorteado` vencedor.

- ❏ Você pode utilizar a classe **java.util.Random** em sua solução.
- ❏ Você pode utilizar a classe **java.util.Arrays** em sua solução para ordenar os números de um array.
- ❏ Para esta implementação, você está livre para utilizar métodos estáticos. Entretanto, o uso inadequado do modificador *static* pode fazer você perder pontos, então pense bem antes de usá-lo.

2. (2,0 - Herança e polimorfismo) Implemente a seguinte hierarquia de classes:

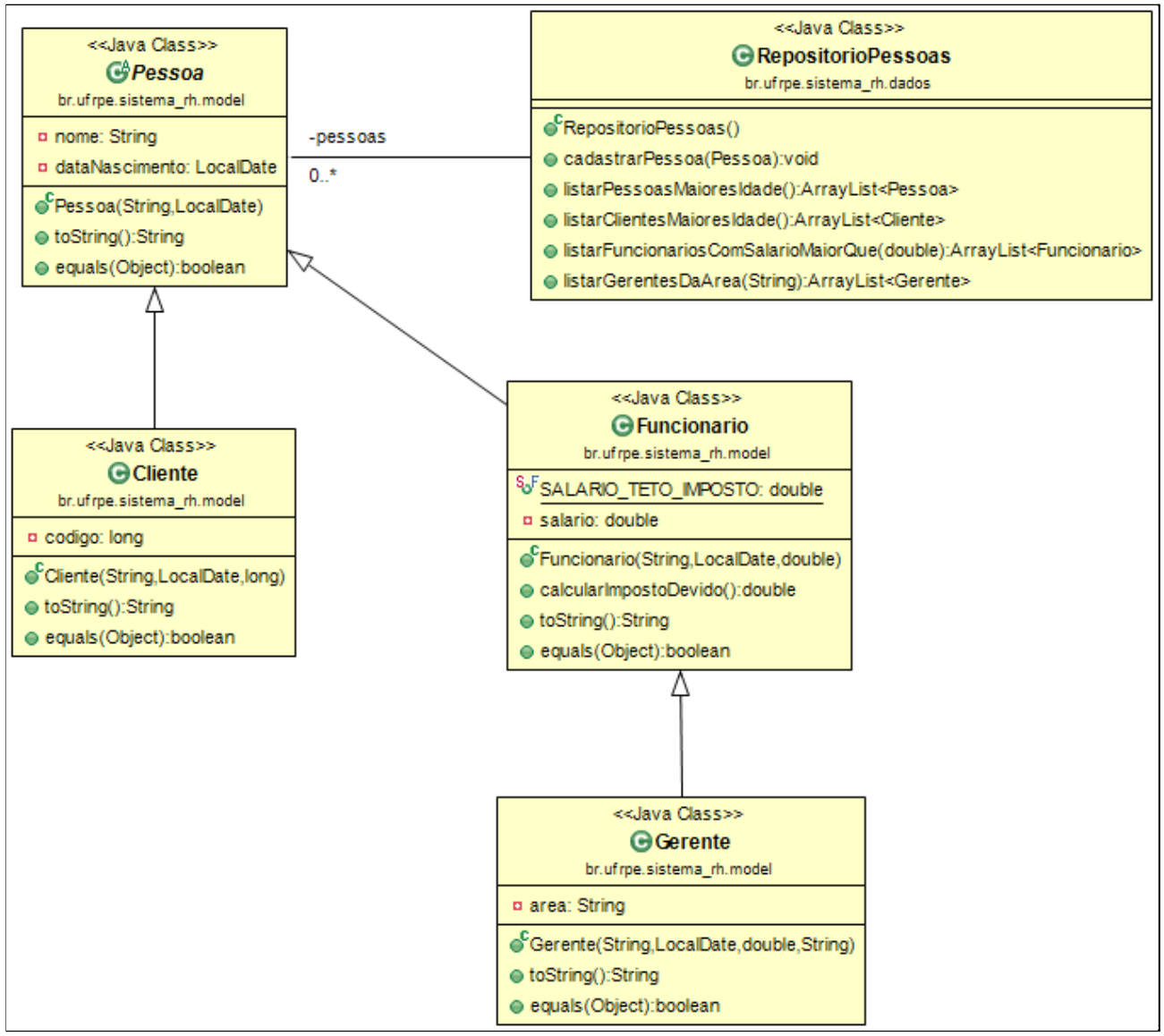

Figura 1. Diagrama de Classes em UML do Sistema de Departamento Pessoal.

Observe atentamente os relacionamentos entre as classes apresentados na Figura 1. Todas as classes apresentam construtores que inicializam todos os atributos. Além disso, todas as classes apresentam o método equals e toString que devem ser implementados sobrescrevendo a assinatura padrão da classe Object e testando se todos os atributos da classe corrente são iguais aos atributos do parâmetro. É importante lembrar que, os métodos equals de classes filhas podem invocar o método equals da classe mãe através de uso da palavra reservada *super*. Todos os métodos get/set foram omitidos no diagrama, mas eles devem existir.

Detalhes das classes:

- Pessoa: **classe abstrata** com construtor que inicializa atributos de acordo com parâmetros.
- Cliente: apresenta um código único e herda de Pessoa. Método toString deve considerar todos os atributos.
- Funcionario: herda de pessoa, apresenta um salário como atributo e uma constante para definir o teto do salário com imposto mais baixo. Também apresenta o método calcularImposto que implementa a seguinte regra:
    - Se o salário for menor ou igual ao SALARIO_TETO_IMPOSTO, o imposto é igual a 5% do salário. Se for maior, o imposto é igual a 7,5% do salário.
    - Método toString deve considerar todos os atributos e, além disso, coletar a informação sobre o imposto através da invocação do método da classe.
- Gerente: herda diretamente de Funcionario e apresenta um atributo que representa a sua área de atuação. Método toString deve considerar todos os atributos.
- RepositorioPessoas: classe que armazena em memória instâncias do tipo Pessoa (nome do atributos é ‘pessoas’, assim como descrito no relacionamento com a classe Pessoa) usando array ou ArrayList, à sua escolha. Construtor inicializa o atributo e deixa a lista vazia. Os métodos processam objetos do tipo Pessoa polimorficamente e fazem o seguinte:
    - cadastrarPessoa(Pessoa): verifica se já existe uma pessoa igual àquela no array ou lista através e, se não houver, insere no final da estrutura. Observe que, para a execução desse método, é necessária a implementação do método equals nas outras classes.
    - listarPessoasMaioresIdade: retorna uma lista de pessoas, de qualquer subtipo, que sejam maiores de idade considerando a data corrente para cálculo da idade.
    - listarClientesMaioresIdade: retorna uma lista de clientes que sejam maiores de idade.
    - listarFuncionariosComSalarioMaiorQue(double salario): retorna uma lista de funcionários que tenham salários maiores do que o informado como parâmetro.
    - listarGerentesDaArea(String area): retorna uma lista de gerentes que sejam da área informada como parâmetro.

Para testar o seu sistema implementa um classe que contenha um método main e execute as seguintes funcionalidades:

➔ Crie uma instância do Repositório de pessoas

➔ Crie cinco instâncias do tipo Cliente e adicione no repositório. Depois de adicioná-las, crie uma outra instância de um cliente, mas que tenha o mesmo código, nome e data de nascimento de algum cliente já criado e tente adicionar ao repositório. Lembre-se

➔ que o repositório não deve permitir adição de instâncias iguais.

- ➔ Execute o mesmo procedimento do passo anterior, porém instanciando funcionários e gerentes. Lembre-se também de instanciar uma última instância igual à alguma já criada e tentar adicionar no repositório.
- ➔ Execute cada um dos métodos do tipo listar* do repositório e, utilizando o retorno de cada um deles, imprima os valores de cada elemento da lista retornada a partir da invocação do método toString correspondente.

3. (2,5 - Classes abstratas e interfaces) Controlador Financeiro

Você foi contratado para fazer um sistema de controle financeiro de uma empresa de software, de forma que seja possível cadastrar todas as transações (receitas e despesas) e fazer o gerenciamento do fluxo de caixa. Para isto, o departamento de engenharia de software forneceu um diagrama UML e algumas informações a respeito dos cálculos a serem considerados e implementados pelo método "calcularTotal()" existente em cada subclasse especificada no diagrama.

Despesas

- As principais despesas da empresa são com internet, água e energia. O consumo é registrado no sistema e o valor da despesa calculado de acordo com as regras a seguir.
- **Água:** calcular com base no consumo em m3 de acordo com as regras abaixo.
    - De 0 a 10 m³ - tarifa mínima comercial R$ 45,00
    - De 11 a 20 m³ - valor por m³ excedente R$ 5,00
    - Acima de 20 m³ - valor por m³ excedente R$ 6,00

Exemplo: para saber o quanto 27 m³ representam em Reais.

1ª faixa. O consumo dos primeiros 10 (dez) m³ de água tem o valor associado a uma tarifa mínima de R$ 45,00.
2ª faixa. O usuário deve multiplicar os outros 10 (dez) m³ pelo valor de excedente na segunda faixa de consumo da tabela acima. (Ex.: 10 m³ X R$ 5,00 = R$ 50,00).
3ª faixa. O usuário deve multiplicar os 07 (sete) m³ restantes pelo valor de excedente na terceira faixa de consumo da tabela abaixo. (Ex.: 7 m³ X R$ 6,00 = R$ 42,00).
4. O sistema deve agora somar os 03 (três) resultados. (Ex.: R$ 45,00 + R$ 50,00 + 42,00 = R$ 137,00).

- **Internet:** calcular com base nos dados trafegados de acordo com as regras abaixo.

Consumo	Valor Fixo	Excedente:
20 GB	R$ 90,90	R$ 3,90/GB adicional
100 GB	R$ 240,90	R$ 4,40/GB adicional
500 GB	R$ 590,90	R$ 5,10/GB adicional

- **Energia:** calcular com base no consumo em kWh de acordo com as regras abaixo.

  Valor do kWh: R$ 0,50

  Cobrança adicional de R$ 1,20 para cada 100 quilowatts-hora (kWh) consumidos

  Adicionar a taxa de iluminação pública:
    - De 0 kWh a 50 kWh: R$ 2,00
    - De 51 kWh a 200 kWh: R$ 15,00
    - Acima de 201 kWh: R$ 35,00

Receitas

- A principal fonte de receita da empresa são as licenças de software que ela mantém. O cálculo é mais simples e consiste em um valor a ser cobrado por usuário (conta) de acordo com o tipo de licença.
    - Licença Básica: R$ 20,00 / conta
    - Licença Plus: R$ 35,00 / conta
    - Licença Enterprise: R$ 150,00 / conta

Programa Controlador Financeiro

Com base nas informações apresentadas no diagrama UML fornecido na Figura 2 a seguir, crie um programa que instancie diferentes despesas e receitas em datas distintas. Adicione todas as receitas e despesas como transações no objeto "FluxoCaixa" (método *adicionarTransacao*) e imprima o resultado da chamada de cada um dos outros métodos que possui retorno. Perceba que o objeto "FluxoCaixa" tratará cada uma das instâncias como sendo do tipo "Transacao" (interface).

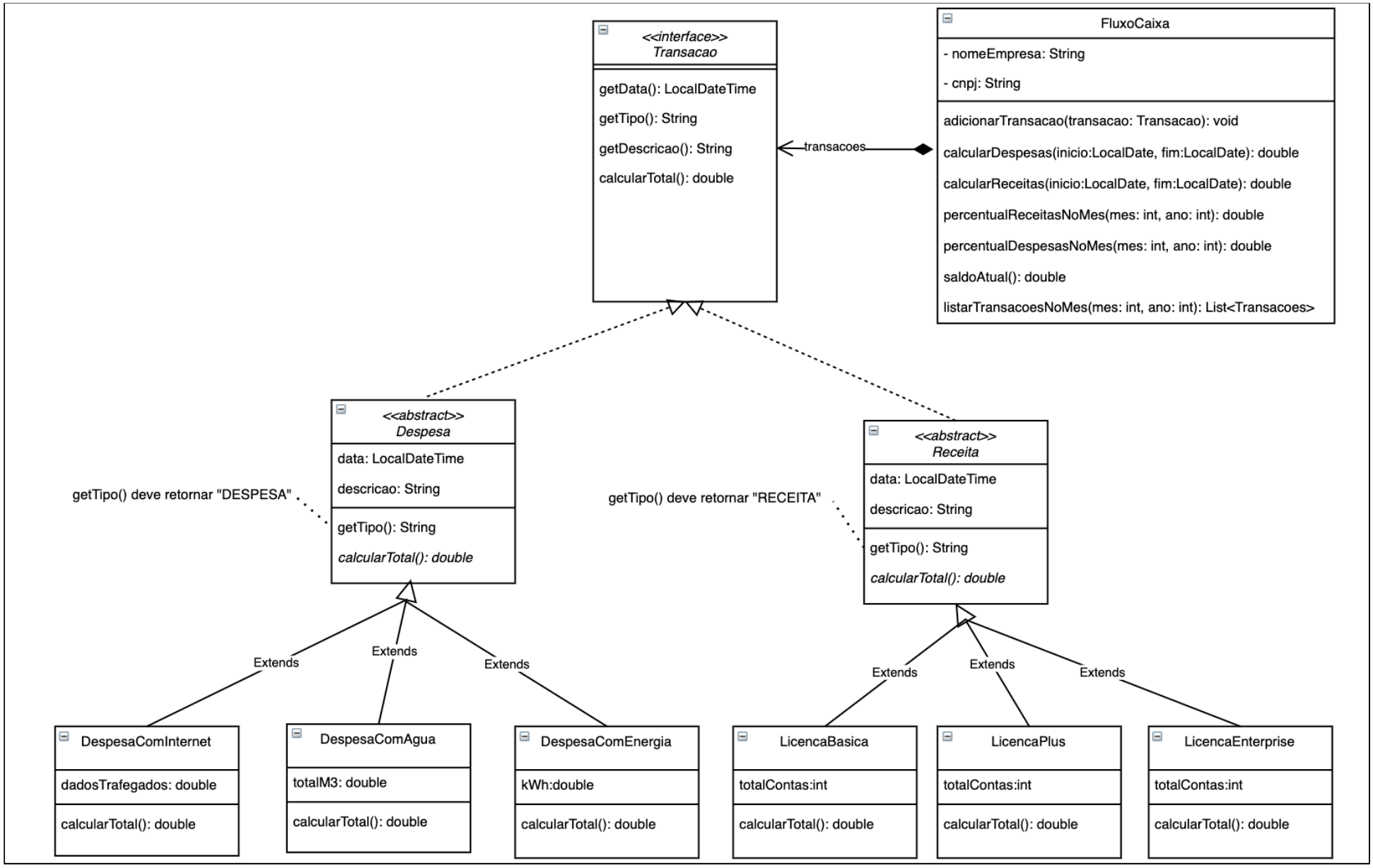

Figura 2. Diagrama de Classes em UML do Controlador Financeiro.

4. (3,5) Mini-sistema comparador de preços

Você foi contratado para implementar um comparador de preços de produtos vendidos em diversos sites de comércio eletrônico. Um exemplo de um serviço como este pode ser encontrado em www.zoom.com.br. Com uma abordagem diferente dos outros exercícios que fizemos até agora, **esta implementação apresenta um conjunto de requisitos e premissas básicas, mas não traz explícitos os relacionamentos entre as classes**.

Os requisitos básicos do sistema são:

- REQ1: O usuário deve ser identificado no sistema através de algum tipo de autenticação, com o objetivo de manter histórico de acesso e conseguir cadastrar alertas de preço (REQ2)
- REQ2: O usuário pode gerenciar (CRUD) alertas de preços para um dado produto. Uma vez que o preço daquele produto foi atingido, o usuário deverá ser notificado.
- REQ3: O sistema deve permitir o gerenciamento de produtos (CRUD) com suas características básicas.
- REQ4: O sistema deve permitir o gerenciamento de lojas (CRUD) com suas características básicas
- REQ5: O sistema deverá permitir a busca pelos preços de um dado produto na data atual, ordenando-os do menor para o maior.
- REQ6: O sistema deverá permitir a apresentação do histórico de preços de um dado produto em um determinado período, com variação de preço por dia

Para melhor entendimento dos requisitos, visite essa busca: https://bit.ly/galaxy_s20_ultra. Observe o histórico de preços e a lista com as possibilidades de ordenação.

a. (0,5) Entidades básicas

Você está livre para definir os tipos de cada atributo, bem como adicionar atributos que atendam melhor à solução proposta. Seguem algumas entidades que minimamente precisarão ser implementadas:

- Usuario: identificador único (e-mail por exemplo), nome, data de nascimento.
- Produto: identificador único, nome, descrição, categoria (este atributo deve ser definido como uma enumeração e você escolher quais categorias seriam adequadas para um produto, tais como, eletrônicos, utensílios domésticos, ferramentas, vestuário, etc).
- Loja: identificador único, nome, endereço eletrônico (site).
- OfertaProduto: esta entidade representa uma oferta de um determinado produto feita por uma loja numa determinada data por um preço qualquer. Em algum momento do seu sistema, manter uma lista de objetos deste tipo vai representar o histórico de preço de um dado produto na linha do tempo.
- AlertaPreco: esta entidade representa o desejo de um objeto do tipo usuário, cuja classe já deve estar definida, ser alertado se um determinado produto atingiu um determinado preço.

b. (0,5) Repositórios genéricos

Para cada uma das entidades mencionadas no ponto ‘a’, você deve instanciar um repositório que mantenha uma lista de objetos daquele tipo. Para evitar duplicação de código, você deve criar um repositório genérico que mantenha um ArrayList de uma entidade genérica e já implemente os métodos básicos de um CRUD, com validações como não permitir inserção duplicada de objetos com o mesmo ID (noção de igualdade necessária - método

equals) e, se houver tentativa de inserção com o mesmo ID, uma exceção informando que o elemento já existe deve ser levantada. Vide exemplo implementado em prova passada aqui: http://bit.ly/repo_generico. O repositório genérico é instanciado nas classes que irão usá-lo (controladores)

c. (2,0) Controlador - funcionalidades e regras de negócio

O controlador deve manter as instâncias de cada repositório do sistema, seguindo a arquitetura MVC já discutida em nossas aulas. O acesso aos métodos dos repositórios deve ser feito através de métodos criados no controlador que delegam (delegate) para os respectivos repositórios. Além disso, o controlador deve ter os seguintes métodos para atender aos requisitos do sistema:

- `List<OfertaProduto> listarOfertasOrdenadasPorPrecoNaData(Produto param, LocalDate dataAtual)`
  Este método atende ao REQ5 e deve implementar a ordenação usando alguma das técnicas já discutidas em aulas (Comparator ou Comparable)
- `Map<LocalDate, List<OfertaProduto>> montarHistoricoDeOfertasDoProdutoNoPeriodo(Produto param, LocalDate dataInicial, LocalDate dataFinal)`
  Este método deve retornar um mapa com todas as datas do período informado como parâmetro associadas a todas as ofertas (preços) que os produtos tiveram em diferentes lojas daquele dia.
- `List<AlertaPreco> verificarAlertasDePrecoAtingido(Usuario u)`
  Este método deve capturar todos os alertas de preço do usuário informado como parâmetro e, para cada alerta, deve verificar se os produtos ofertados na data atual tiveram algum preço atingido (igual ou menor que o alerta). A lista de alerta de preço atingido deve ser retornada.

d. (0,5) Classe Principal contendo método main para testes

A existência dessa classe e do método main é primordial para a execução e teste do mini-sistema comparador de preços. Sem ela, as validações do funcionamento do sistema ficam inviáveis. Para um método main, você deve:

- Criar pelo menos 1 usuário e salvá-lo no respectivo repositório através do controlador
- Criar pelo menos 2 produtos e salvá-los no respectivo repositório através do controlador
- Criar pelo menos 2 alertas (1 para cada produto) e salvá-los no respectivo repositório através do controlador
- Criar pelo menos 3 lojas e salvá-las no respectivo repositório através do controlador
- Criar pelo menos 12 ofertas de produtos (4 para cada loja) feitas em datas distintas e salvá-las no respectivo repositório através do controlador

Uma vez criados os objetos e salvos nos repositórios, você deve invocar os métodos listados no ponto ‘c’ verificando seu correto funcionamento. Para o método `listarOfertasOrdenadasPorPrecoNaData`, você deve receber seu retorno e imprimir a loja | preço | endereço da loja por linha (tal qual o site zoom). Para o método `montarHistoricoDeOfertasDoProdutoNoPeriodo`, você deve receber seu retorno e imprimir a data ordenada de menor para maior | menor preço atingido na data (simulação de um gráfico de linha onde o eixo Y é o menor preço e eixo X é a linha do tempo com as datas ordenadas). Para o método `verificarAlertasDePrecoAtingido`, você deve receber seu retorno e, para cada objeto retornado na lista, apresentar um Alert na tela (consulte em https://bit.ly/2QbYwH2).

HAPPY CODING!!!