**UFCG -** Universidade Federal de Campina Grande

**Ciência da Computação**

**Eduardo Henrique Pontes Silva**

**Gustavo Luiz Bispo dos Santos**

**João Pedro de Barros Barbosa**

**Rafael Dantas Santos de Azevedo**

**Relatório do Projeto de Laboratório - "**~~Lista pra mim~~©"

Campina Grande

2018

# Atividades Desenvolvidas

* 1. Caso 1:

No caso 1 foi feita a classe abstrata *Item*, que possui os atributos mais básicos de um item como nome e categoria, ambas Strings, um inteiro id(identificador único) e um atributo que guarda o menor preço, em double. Além disso, tomamos a decisão de guardar os preços do itens em um mapa, que associa o estabelecimento (chave) que vende o item, a um preço.

Aplicamos herança para os diferentes tipos de item, que possui a classe abstrata *item* como pai e as classes filhas *ItemPorUnidade*, *ItemPorQuantidadeFixa* e *ItemPorQuilo*. Por fim, foram criados todos os métodos necessários, de acordo com os testes de aceitação, e um *controllerItem*, que é responsável por fazer toda a lógica de Item e guarda um mapa de itens que relaciona o id (chave) a um Item.

* 1. Caso 2:

No caso 2 foram criadas duas classes especializadas em fazer comparações de *Item*, onde ambas implementam a interface *Comparator*.

A primeira, compara as suas representações em String; a segunda, compara os menores preços dos itens. Além disso, no *ControllerItem*, foram criados os métodos "getItem" que permite a listagem de itens, ordenados lexicograficamente; "getItemPorCategoria" que permite a listagem de itens de uma dada categoria, ordenando lexicograficamente; "getItemPorMenorPreco" que lista os itens de acordo com o menor preço e "getItemPorPesquisa" que retorna todos os itens relacionados a uma String passada, ordenados também lexicograficamente. Em todos os métodos, basicamente adicionamos todos os itens necessários a um *ArrayList* e utilizamos o *collections.Sort* junto com o comparador adequado para fazer essa ordenação.

* 1. Caso 3:

No caso 3 como medida de design para facilitar a ordenação das categorias citadas como default, transformamos a String categoria em um *ENUM Categoria*, que possui como constante as categorias que os itens podem se enquadrar. Além disso criamos a classe *ListaDeCompras* que é identificado pelo seu nome (descritor) e contém um *HashSet* de compras. Decidiu-se usar o *HashSet* para que não hajam compras repetidas.

Também foi criada a classe *Compra* que contém como atributos Item e quantidade. Tomamos essa medida pois em uma lista de compras se armazena compras, e um item só é uma compra quando possui uma quantidade associada a ele. Por Fim, criou se um *ControllerLista* que recebe o *ControllerItem* como parâmetro para que fosse possível fazer as ligações entre os controllers. Este também faz toda a lógica necessária para as listas de compras e guarda um mapa de listas de compras que associa o nome da lista (chave) a uma lista de compras.

* 1. Caso 4:

No caso 4 foi criada uma lógica no *ControllerLista* para a pesquisa de listas de compras. Além disso, foram criados os métodos privados "getListasDoDia", que retorna uma lista com as listas criadas em uma data específica, e o "getListasPorItem", que retorna uma lista com as listas de compras que possuem o item passado. Com esses dois métodos privados é possível que toda a lógica do caso quatro funcione.

* 1. Caso 5:

No caso 5, para o funcionamento do método que gera uma lista automática a partir da última lista foi necessário criar um comparador de tempo, que também implementa a interface *Comparator*. A partir do *Collections.Sort*, se acha a última lista criada e adiciona todos os itens dela a lista automática 1. Na geração da segunda lista automática, usa-se uma tática semelhante, a diferença é que é feita a cópia da última lista que possui o item passado como parâmetro, para isso foi adicionado um ***for*** que percorre todas as listas criadas ordenadas por datas, verificando se há o item na lista, caso haja, copia-se as compras desta lista.

Na terceira Lista automática, foi usada uma tática completamente diferente, pois é necessário que se crie uma lista com os itens mais presentes em outras listas já criadas. Para tal, foi necessário percorrer todos os itens e verificar se eles estão presentes em mais de 50% da listas e, em caso afirmativo, é gerada uma compra a partir do item com a média das quantidades das outras listas e é adicionado à lista automática 3.

* 1. Caso 6:

No caso 6, adicionamos uma classe que sugere o melhores estabelecimentos para compras. Para isso foi criado um método privado que cria listas temporárias contendo listas de compras geradas a partir dos locais que possuem os itens da lista passada. Para a criação desta listas temporária foi necessário criar um método dentro de item que passa todos os estabelecimentos que possuem o item. Com esses estabelecimento, o método cria listas de compras com os nomes do estabelecimento onde são adicionados os itens que estão presentes tanto na lista parâmetro como no estabelecimento. O preço sugerido é calculado a partir da quantidade e preços dos itens das compras adicionadas à lista.

* 1. Caso 7:

No caso 7 criamos uma classe *Persistencia* com os métodos "iniciaSistema" e "fechaSistema". Essa classe armazena todas as coleções dos controllers e o id do último item cadastrado. Para que as escrita fosse possível, fez-se necessário que os objetos que tem relação com as coleções salvas implementassem a interface Serializable. Além disso a escrita de todos os dados é feita em um arquivo único. No caso da leitura, esta é feita na mesma sequência em que os dados foram gravados, estes são atribuídos às coleções e ao id de item no sistema.

* 1. Métodos auxiliares "*pegaItem"*:

Para auxiliar na captura de itens pelo controller lista sem quebrar o acoplamento e as noções de expert da informação, foi necessário criar dois métodos públicos, onde recebe-se o id do item e uma mensagem que pode ser usada, caso se obtenha um erro, e o segundo sendo semelhante ao primeiro, exceto por não receber a mensagem de erro.

# Divisão das Atividades Desenvolvidas

**Eduardo Henrique :** Implementação Facade e controller, implementação do caso 2, implementação caso 5, auxílio caso 6, testes Junit de comparadores, criação do diagrama, relatório ;

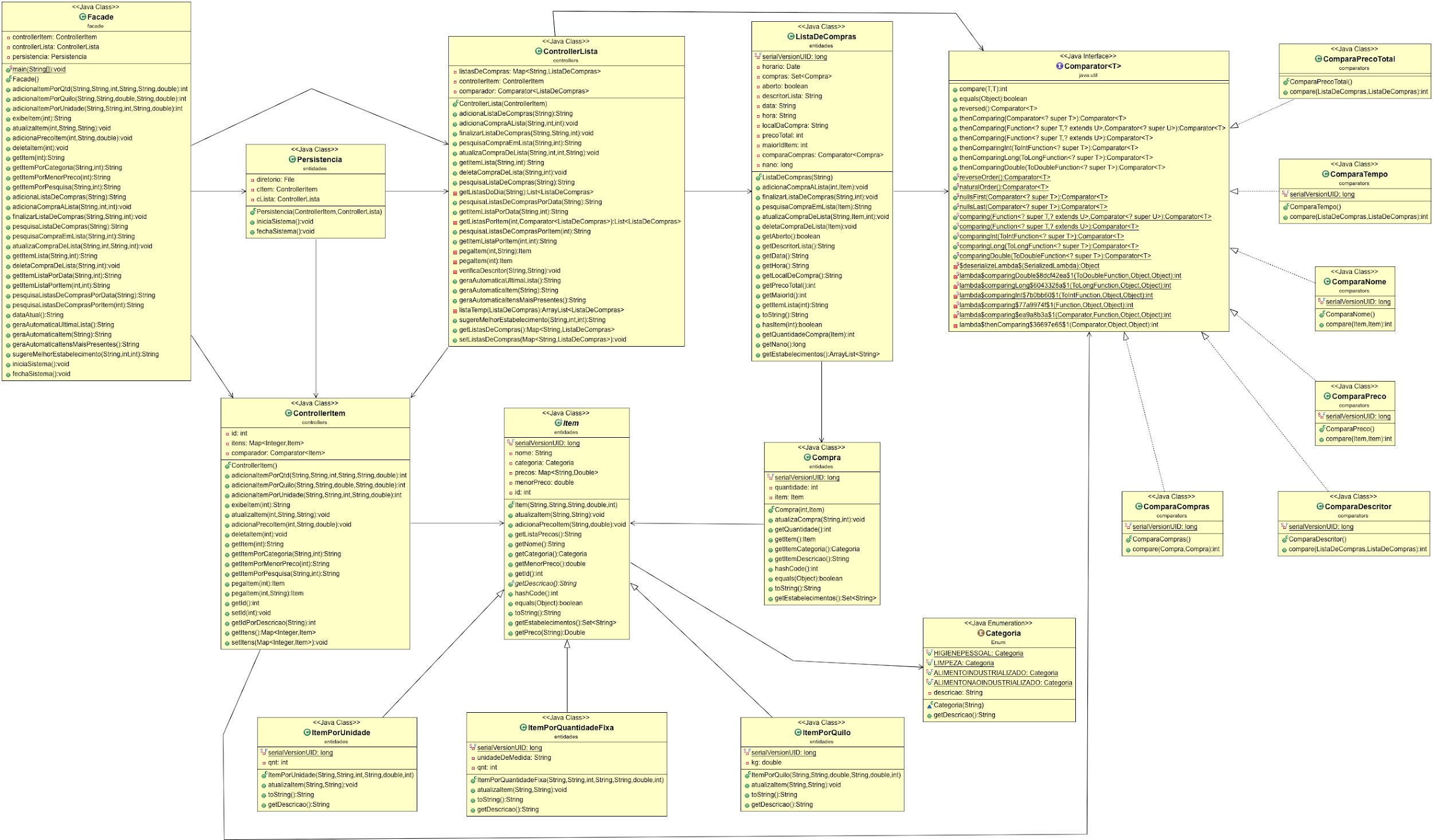
**Gustavo Santos :** Implementação caso 1, implementação do caso 3, implementação do caso 7, testes JUnit dos casos implementados, revisão de código, relatório.

**João Pedro de Barros:** implementação do caso 1, implementação do caso 4, implementação do caso 6, revisão de código, testes Junit, tratamento de exceções.

**Rafael Dantas:** implementação do caso 2 (criação das classes comparadoras), implementação do caso 6, revisão de código, documentação do código.

# Design e Diagrama de classes

Escolhemos este Design visando o melhor acoplamento e definição de responsabilidade. Criamos as classes *Item* e suas filhas (*ItemPorUnidade*, *ItemPorQuantidadeFixa* e *ItemPorQuilo)*; *ListaDeCompras* e *Compra*, onde todas são expert da informação que guardam. O Creator de itens é o *ControllerItem*, o de listas é o *ControllerLista*, o de *Compra* é *ListaDeCompra*. Além disso temos o ENUM Categoria, que guardas constantes que representam as categorias de itens.

****

Por fim, há também as classes que implementam a interface *Comparator* e que possuem a capacidade de fazer comparações entre objetos. As classes *ComparaNome* e *ComparaPreco* possuem a capacidade de fazer comparações entre objetos do tipo *Item* e seus subtipos filhos (*ItemPorUnidade*, *ItemPorQuantidadeFixa* e *ItemPorQuilo)*; as classes *ComparaDescritor*, *ComparaTempo* e *ComparaPrecoTotal* fazem comparações de objetos do tipo *ListaDeCompra*; e a classe *ComparaCompra* tem a capacidade de fazer comparações de objetos do tipo *Compra*.

# Conclusão

Diante do tempo gasto e do conhecimento exercido e adquirido, conclui-se que o desenvolvimento deste projeto foi de suma importância para exercitar tudo aquilo que foi aprendido durante as disciplinas e o trabalho em equipe. Além disso, pode-se também trabalhar a pesquisa no que diz respeito às várias frentes possíveis de resolver um mesmo problema.