Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática – CEEI
Unidade Acadêmica de Sistemas e Computação – UASC
Disciplina: Laboratório de Programação 2

Laboratório 05

Como usar esse guia:

- Leia atentamente cada etapa
- Referências bibliográficas incluem:
  - material de referência desenvolvido por professores de p2/lp2 em semestres anteriores (ONLINE)
  - o livro Use a cabeça, Java (LIVRO-UseCabecaJava)
  - o livro Java para Iniciantes (Livro-JavaIniciantes)
- Quadros com dicas tem leitura opcional, use-os conforme achar necessário

Sumário

Acompanhe o seu aprendizado

Conteúdo sendo exercitado

- Leitura e entendimento de código e design feito por terceiros
- Evolução de software
- Testes de unidade
- Design, com ênfase em na atribuição de responsabilidades (tipicamente responsabilidade única), controladores e coesão
- Uso de coleções
- Uso de interfaces

Objetivos de aprendizagem

Temos dois objetivos principais nesse lab:

1. ter experiência de um lab parecido com um projeto em termos de tamanho;
2. experimentar com entendimento e evolução de um design feito por outra pessoa e envolvendo uso de interfaces e coleções.

Para se aprofundar mais...

- Referências bibliográficas incluem:
    - material de referência desenvolvido por professores de p2/lp2 em semestres anteriores (ONLINE)
    - o livro Use a cabeça, Java (LIVRO-UseCabecaJava)
    - o livro Java para Iniciantes (Livro-JavaIniciantes)
- Projetando com interfaces
- Um pouco mais sobre GRASP

Introdução

Neste laboratório queremos experimentar como entender e estender um software OO. Partiremos de um sistema pronto e iremos introduzir novas funcionalidades nesse sistema e garantir que o sistema esteja correto após as modificações. Vamos exercitar a leitura e extensão de código escrito por outra pessoa.

O uso de testes automáticos é essencial para evoluirmos com um código grande. É a única forma prática de conferirmos que um grande número de funcionalidades ainda está OK depois de alguma modificação. Um ponto muito importante desse laboratório é que **usaremos os testes para entender o que você implementou.** Leremos os testes para entender o código. O que não estiver testado e compreensível nos testes não será

considerado. Por isso, é melhor ir fazendo os testes na medida que você vai implementando as alterações. Não deixe para fazer os testes no final.

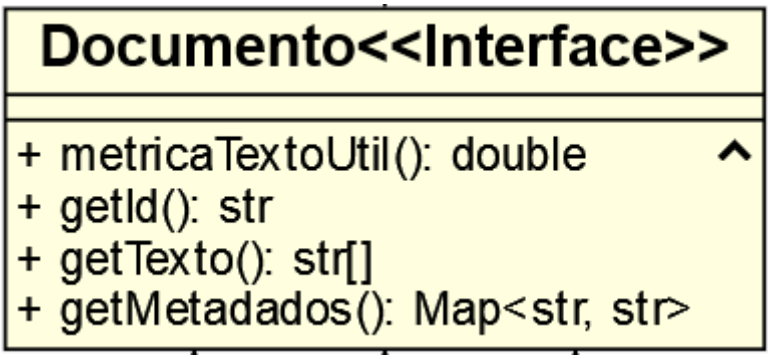
O sistema que evoluiremos: **Lunr 0.1**

Lojas de e-commerce, de registro de eventos, cartórios e controles de processos são sistemas que operam no cadastro de dados e na busca ou identificação de tais dados dentro de uma base de dados.

Por exemplo, em um e-commerce é importante cadastrar produtos e ser capaz de pesquisar por tais produtos. Essa busca deve procurar, por exemplo, termos de pesquisa que fazem parte da descrição do produto, como também deve ser possível procurar por metadados, como a marca do produto entre outros. No e-commerce deve ser possível procurar produtos semelhantes ao que está sendo comprado. No sistema de gestão dos códigos dos alunos em P2, podemos cadastrar códigos, buscar por códigos que usem determinada estrutura de código, e identificar plágios.

Vários sistemas apresentam essa estrutura: o cadastro de dados textuais agregados a metadados e que possuem uma identificação única. De forma genérica, podemos chamar essa estrutura de **DOCUMENTO**. Ao criar uma generalização desta natureza, podemos criar um sistema de gerência de documento que possa ser útil em diferentes cenários distintos. **Este é o Lunr: um sistema de controle e busca de documentos**.

O documento é uma estrutura com as seguintes operações:



Onde teremos os métodos:

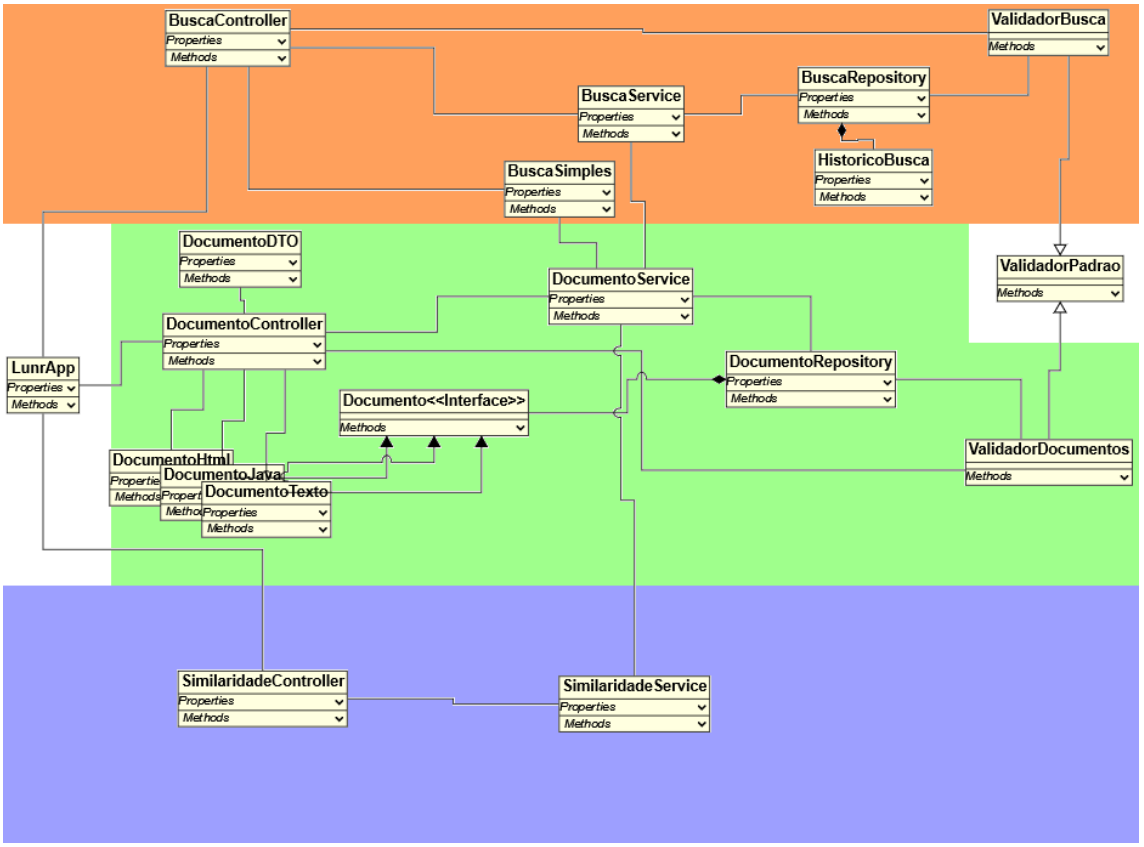
- metricaTextoUtil: retorna uma proporção referente à quantidade de texto útil (textos com termos que são expressivos ou de interesse para o sistema) sobre o total de caracteres originais no documento
- getId: retorna a identificação do documento
- getTexto: retorna um array de termos não-vazios que são relevantes do documento
- getMetadados: retorna um mapa de metadados, ou seja, informações úteis sobre o documento mas que não estão, necessariamente, dentro do texto do

documento. A chave descreve sobre o que se refere o metadado em questão (ex.: "AUTOR") enquanto o valor é a representação em string do metadado em si (ex.: "Matheus G").

Observe que essa abstração permite cadastrar diferentes tipos de dados! Poderíamos criar um Documento que represente um Produto, outro que represente uma Prova, um Código Java, uma Página da Internet, etc. Um sistema de busca, apresentação ou similaridade pode operar sobre essa entidade, desde que o documento implementado ofereça os métodos em destaque.

Neste laboratório, já implementamos a base do Lunr. Você baixará o código inicial do sistema aqui. O sistema foi implementado com 3 funcionalidades, onde cada funcionalidade está associada a um pacote distinto, e com as classes de cada funcionalidade identificadas no UML abaixo:

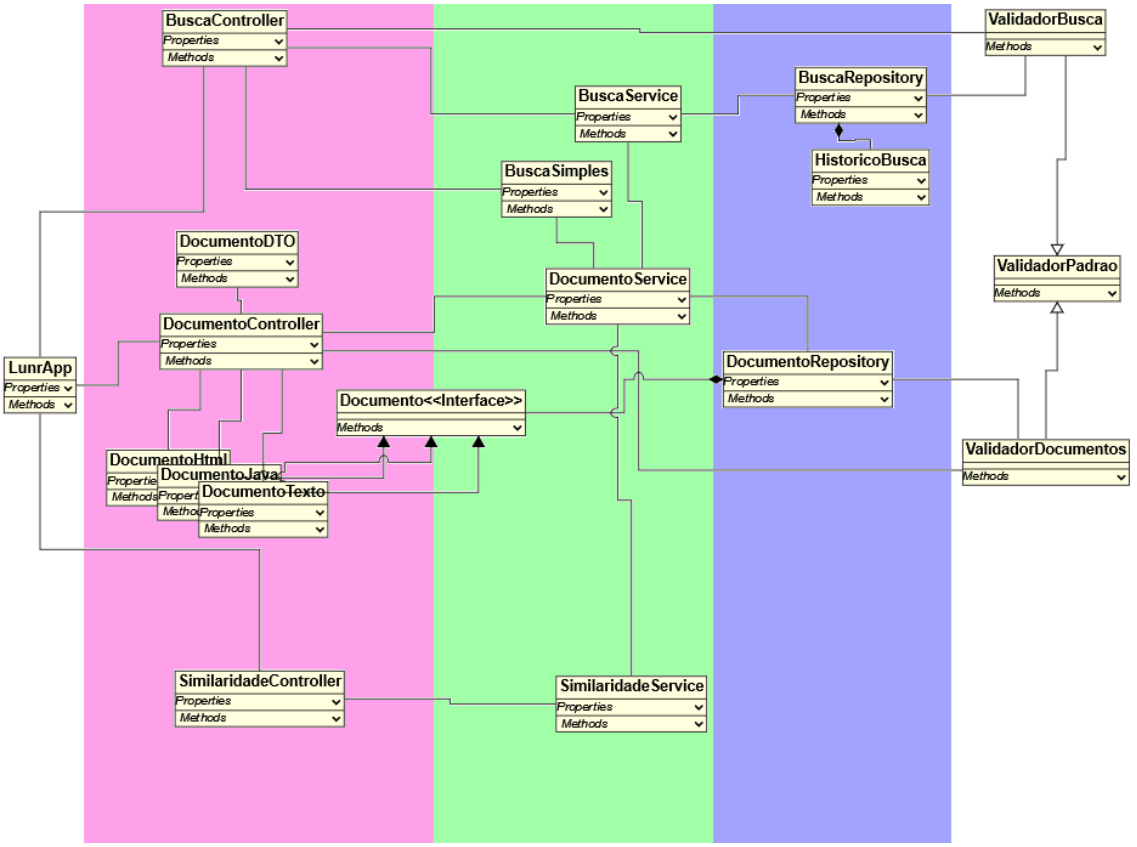
- Busca: Subsistema de busca. Ao contrário dos documentos, o subsistema de buscas trata de buscas mais complexas, definir ordenação, ranking, relevância, dado as consultas realizadas no subsistema de documentos
- Documentos: Subsistema de gerência de cadastro e recuperação de documentos. Esse código é responsável por algum nível de busca / acesso de documentos simples.
- Similaridade: Subsistema para realizar os cálculos de similaridade de documentos.



A divisão de funcionalidades por pacotes é uma prática comum em sistemas avançados pois permite delimitar o acesso de um subsistema a outro, bem como a rápida identificação de código associado a uma determinada funcionalidade.

Do ponto de vista estrutural, o sistema é separado em 3 camadas (da esquerda para a direita):

- Controller/UI: Classes de interação com o usuário ou com requisições do usuário, fazendo validação básica e propagando essas requisições para as classes resposáveis pela lógica do sistema.
- Services/Lógica de Negócio: Classes responsáveis pela lógica do sistema. Manipulam entidades e elementos que são próprios do sistema.
- Repository/Dados: Entidades responsáveis pelo armazenamento e operações de acesso aos dados.



As classes DocumentoHTML, DocumentoJava, DocumentoTexto têm uma caracterização ambígua nesse sistema. Elas, em si, são classes de dados, e estariam mais próximas na camada roxa, mas estão sendo usadas aqui mais como parte do tratamento da entrada do usuário / sistema para que a lógica de negócio trabalhe apenas com o tipo Documento.

No UML da organização das classes é possível visualizar também entidades que não estão descritas especificamente em nenhuma camada:

- LunrApp: responsável por inicializar as principais classes do sistema (Controllers e Services) e realizar a ligação entre estas classes.
- Validadores: responsáveis por fazer validação de entrada. São utilizados tanto por controllers (para validar o que chega do usuário) como por repositórios para garantir a consistência das entidades a serem armazenadas.

Em sistemas corporativos[1] como o Lunr, de fato, tanto a inicialização dos componentes, como a estratégia de validação é feita de forma implícita de acordo com o sistema implementado. Aqui, expomos as classes e suas ligações para evitar apresentar conceitos que são mais avançados para programação OO.

[1]Sistemas de maior porte das empresas e com grande escopo de funcionalidades

No sistema foram implementados apenas alguns testes funcionais. São testes que avaliam apenas a macro-funcionalidade do código, sem explorar ou se importar com os detalhes internos das classes ou situações de fluxo alternativo de código, ou seja, não foram implementados testes de unidade.

Parte 1: Estrutura de Documentação

Sua primeira atividade é melhorar o design do código fornecido através do refatoramento da classe DocumentoRepository de modo a escolher as melhores coleções para armazenar e manipular os documentos.

Ao pensarmos na escolha de uma coleção muitas vezes o primeiro ponto que vem a nossa mente é a facilidade de uso, mas diversos outros aspectos podem ser considerados. Considere por exemplo, situações onde temos uma grande quantidade de dados sendo frequentemente acessados.

Nesse tipo de situação é interessante pensar no desempenho do sistema e com isso considerarmos coleções que permitam uma busca e leitura/escrita mais rápidos, mesmo apresentando maior complexidade de uso.

Lembre-se: refatorar é alterar a estrutura interna do código, incluindo: criar ou remover classes, métodos e atributos, mas MANTENDO a mesma funcionalidade que já existia antes.

Observação: ao analisarem a estrutura de documentos com atenção, perceberão que existe redundância de código e este seria um bom ponto de refatoramento também. Porém, essa não é sua tarefa neste momento ou neste laboratório :)

Parte 2: Evolução do sistema

Evolua o Lunr versão 0.1 para a versão 1.0 e permita que ele seja lançado! Para isto, você deve implementar três casos de uso (em inglês, Use Cases, US) adicionais.

US1: Busca Avançada

A busca simples, que já está implementada, é determinada por um conjunto de termos que devem ser procurados entre os documentos armazenados no sistema. Na busca simples, caso um documento tenha pelo menos 1 dos termos de busca, ele se torna resultado da busca e os documentos com presença de mais termos diferentes da busca se tornam mais relevantes. A quantidade de vezes que cada termo aparece na pesquisa é irrelevante.

Na busca avançada os documentos são procurados a partir dos seus metadados. O subsistema de busca deve receber um mapa de metadados a serem pesquisados.

Exemplos de entradas desse mapa: : <"TIPO", "txt"> e <"LINHAS", "1">. A busca avançada deve selecionar TODOS os documentos que tenham TODOS OS METADADOS indicados. Não é necessário ordenar os resultados retornados, nem limitar a quantidade de respostas.

Desenvolva um sistema extensível, que será capaz de receber outros tipos de busca. Faça passar (e ajeite, se necessário) os testes de busca avançada já existentes.

US2: Análise de Similaridade

Como todos os documentos podem ser descritos pelos termos que fazem parte de seus textos, é possível fazer uso dessa informação para calcular a similaridade entre os documentos.

Existem mais de 100 estratégias diferentes para o cálculo de similaridade entre 2 elementos. Para a nossa primeira versão, vamos usar uma das estratégias mais comuns de similaridade, que é calculada pelo coeficiente de Jaccard.

O coeficiente de Jaccard é calculado entre dois conjuntos A e B como o tamanho da interseção destes conjuntos sobre o tamanho da união desses conjuntos. Ou seja, dado os conjuntos A e B, a distância J é calculada como:

$$J = \frac{|A \cap B|}{|A \cup B|}$$

Por exemplo, um documento é gerado da frase "Uma casa feliz é uma casa bonita". Desse documento podemos extrair o conjunto de termos A = {Uma, casa, feliz, é, bonita, uma}. Do documento gerado pela frase "Um dia feliz é um bom dia", nós temos os termos B = {Um, dia, feliz, é, um, bom}.

A similaridade entre os dois grupos então é:

|{feliz, é}| / |{Uma, casa, feliz, é, bonita, uma, Um, dia, um, bom}|

Ou seja, 2 / 10, ou 20% de similaridade.

Implemente o método de cálculo de similaridade. Inclua testes associados a essa pesquisa.

US3: Apresentação de Documentos

Em vários sistemas é comum que o usuário possa definir maneiras diferentes de apresentar o conteúdo de arquivos.
Em arquivos Java ou HTML é interessante, por exemplo, poder ver as primeiras linhas para identificarmos o autor e algumas indicações/marcações gerais que nos ajudam a entender o sistema (pacotes, importações, codificação, etc).

Outras opções como ver as últimas linhas de um arquivo de log ajudam para que consigamos encontrar mais rapidamente os problemas mais recentes que aconteceram no sistema, sem a necessidade de "rolar" centenas ou até mesmo milhares de linhas até chegar no ponto que queremos.

Implemente uma solução extensível para permitir diversas maneiras de apresentar o conteúdo de um arquivo. Inicialmente considere três possibilidades:

- Imprimir as primeiras n linhas;
- Imprimir as últimas n linhas;
- imprimir o conteúdo todo em caixa alta.

Extras - Testes adicionais

Como adicional, você deve adicionar testes funcionais e de alto nível (e corrigir, se necessário), aos testes já existentes das funcionalidades indicadas abaixo:

- totalDocumentos
- concatena
- sumariza

Você deve também criar testes de unidade para as classes:

- DocumentoService
- DocumentoRepository
- Classes de dados de documentos (como DocumentoHTML)
- BuscaService
- BuscaRepository
- Classes de lógica de negócio de busca (como BuscaSimples)

Sobre a entrega

Você está recebendo um código que tem testes e onde todos passam. Você deve entregar um código que tem mais testes que todos passam e que permitem que entendamos as modificações que você fez no projeto lendo os testes. Ou seja, tudo que você mudar deve ser testado. Para o que você mudar que precise de alterações nos testes que passamos, altere-os e atualize-os.

Considerando isso, faça um **zip** da pasta do seu projeto. Coloque o nome do projeto para:

- LAB5_SEU_NOME e o nome do zip para LAB5_SEU_NOME.zip. Exemplo de projeto: LAB5_LIVIA_SAMPAIO.zip. Este zip deve ser submetido pelo Canvas.

Seu programa será avaliado pela corretude, pela facilidade de compreendermos o que você fez a partir dos testes, e pelo DESIGN do sistema.