T2 ALEST I

📎

- **Disciplina:** Algorítmos e Estrutura de Dados I
- **Professora:** Isabel Harb Manssour
- **Alunos:** Gabriel Guinter Herter, Gabriel Wagner Piazenski

Introdução

Nesse trabalho foi proposto uma solução para uma calculadora de expressões matemáticas capaz de realizar operações com parênteses, colchetes e chaves. Apenas as seguintes operações serão consideradas: soma(+), subtração(-), multiplicação(*), divisão(/) e potência(^).

Para implementar a calculadora, foram realizados alguns divisões de responsabilidades dentro do código. Nossa implementação utilizou de alguns conceitos de camadas de abstração relacionados aos conceitos de arquitetura limpa. A camada de infraestrutura com o uso de interfaces para remover a dependência do leitor de arquivos foi desconsiderada para manter os padrões disponibilizado pela professora.

Descrição

O algoritmo foi desenvolvido em 4 camadas de abstração:

➡️ Delivery

Nessa camada temos toda a lógica relacionada as camadas externas do projeto.

Temos uma pasta `handler` e uma pasta `builder`.

- `builder`
  - Dentro dessa pasta temos a classe `ResponseBuilderCli` responsável por montar as respostas de sucesso e de erro retornadas para o usuário em linha de comando
- `handler`
  - *Dentro desta pasta temos a classe `FileReaderCalculatorCliHandler` com o método `execute`, que recebe o caminho para acesso ao diretório do arquivo `.txt` onde encontram-se a expressões. Esse método é responsável por executar o leitor de arquivos, instanciar a classe `ExpressionDto` , a qual será detalhada posteriormente e, executar o método estático `CalculatorController.execute` passando como parâmetro um objeto de `ExperssionDto`. Com a resposta proveniente do `CalculatorController.execute` executamos o método `ResponseBuilderCli.buildSuccess` para montar a resposta, a qual, imprimimos na linha de comando.*
  - Em caso de erros do tipo `InvalidExpressionException` executamos o método `ResponseBuilderCli.buildError` e imprimimos sua resposta na linha de comando.

➡️ Controller

Nessa camada temos a classe `CalculatorController`. Essa classe tem a responsabilidade de converter as expressões em um objeto conhecido e já validado para em seguida executar nosso caso. Isso ocorre com a execução do método:

- `execute(ExpressionDto expressionDto)`
  - Esse método tem a responsabilidade de instanciar um `ExpressionModel` com o `ExpressionDto` recebido pelo método e, em seguida chamar o método `CalculateExpressionUseCase.execute` executando nosso caso de uso.

➡️ Use Cases

Nessa camada temos a orquestração do nosso domínio, chegando em um resultado para nossa expressão. Isso ocorre com o uso da classe `CalculateExpressionUseCase` e seu método:

- `execute(ExpressionModel expressionModel)`
  - Esse método executa o método `ExpressionSimplifierService.simplify` até que a expressão esteja resolvida, para isso utiliza de seu método privado `isSolved`, que recebe uma expressão em forma de pilha e verifica se a mesma foi resolvida por completo.
  - O resultado do final é convertido de Pilha para String e em seguida convertido em Double para que seja retornado.

➡️ Domain

Nessa camada temos o coração de nossa aplicação em uma estrutura dividida por diversas pastas:

- `calculator`
  - Nessa pasta temos a classe `Calculator` responsável por calcular operações entre dois operandos e um operador.
  - Nessa classe temos o método calculate que recebe um `Double firstOperand`, um `Operator operator` (Enum que será explicado em seguida) e um `Double secondOperand`. Dependendo do `operator` realizamos uma operação diferente entre o `firstOperand` e `secondOperand` .
- `dto`
  - Essa pasta contém a classe `ExpressionDto`, o qual é o nosso objeto de transferência de dados.
  - Nessa classe recebemos uma `String` com a expressão, a validamos utilizando o método `ExpressionDtoValidation.validate` e atribuímos a pilha representante da expressão recebida em um atributo privado `expression`. Essa pilha que representa a expressão recebida como `String` é o retorno dado pelo método `ExpressionDtoValidation.validate` em caso de sucesso.
- `enums`
  - Essa pasta contém o enum `Operator` criado para representar as possíveis operações a serem realizadas na aplicação. `Addition`, `Multiplication`, `Division` , `Exponential` e `Subtraction`.
- `exceptions`
  - Essa pasta contém as excessões personalizadas da nossa aplicação.
  - `InvalidExpressionException`
    - Nessa classe estendemos a classe `Exception`, recebemos uma `String message` e `String expression` em seu construtor. `message` é repassado para o construtor da classe pai, e `expression` é atribuído ao atributo privado `expression`.
- `model`
  - Nessa pasta temos a classe `ExpressionModel` responsável por representar o nosso objeto de expressão validado e pronto para ser processado.
  - `ExpressionModel`
    - Recebe em seu construtor um `ExpressionDto expressionDto`, e com ele atribui o valor de expression ao seu atributo privado `expression`.
    - Essa classe tem um atributo privado `Double result` que pode ter um valor atribuído somente uma vez. Esse é o resultado da nossa expressão.
- `services`
  - Nessa pasta temos nosso serviço de simplificação de expressões, o qual orquestra as outras classes do nosso domínio para obter uma simplificação da expressão recebida.
  - `ExpressionSimplifierService`
    - Essa classe tem o método `simplify` que recebe uma expressão em forma de pilha. Esse método orquestra outros 12 métodos auxiliares para simplificar a expressão.
    - `public static Double isNumber(char element)`
      - Retorna o numero representado por `element` ou null.
    - `public static boolean isOpen(char element)`
      - Retorna true se `element` for um dos caracteres de "abre".
    - `public static boolean isClose(char element)`
      - Retorna true se `element` for um dos caracteres de "fecha".

        - `public static Operator isOperator(char element)`
            - Retorna um `Operator` caso element seja um operador.
        - `private static boolean isSpace(char element)`
            - Retorna `true` se element for um espaço.
        - `private static void processSpace()`
            - Realiza as operações apropriadas para quando `element` for um espaço.
        - `private static void processOperator(Operator element)`
            - Realiza as operações apropriadas para quando `element` for um operador.
        - `private static void processNumber(Double number)`
            - Realiza as operações apropriadas para quando `element` for um numero.
        - `private static void processOpen()`
            - Realiza as operações apropriadas para quando `element` for um "abre”.
        - `private static void processClose()`
            - Realiza as operações apropriadas para quando `element` for um "fecha”.
        - `private static void processDecimal()`
            - Realiza as operações apropriadas para quando `element` for um decimal.
        - `private static void processNegativeNumber()`
            - Realiza as operações apropriadas para quando `element` for um numero negativo.
- `types`
    - Nessa pasta temos a classe Stack (Pilha). A mesma contém os métodos básicos:
        - `public Character pop()`
        - `public Character top()`
        - `public int size()`
        - `public boolean isEmpty()`
        - `public void clear()`
    - Foram adicionados os seguintes métodos auxiliares:
        - `public static Stack clone(Stack stack)`
            - Retorna uma cópia exata da pilha recebida.
        - `public String toString()`
            - Retorna uma cópia exata da pilha recebida como `String`.
- `validation`
    - Nessa pasta temos a classe `ExpressionDtoValidation`, responsável pela validação da classe `ExpressionDto`.
    - A validação ocorre por meio do método `validate` o qual recebe uma `String` com a expressão e retorna uma pilha igual a expressão caso a mesma seja válida.
    - Caso a expressão não seja válida uma excessão `InvalidExpressionException` é lançada.
    - Para a validação são utilizados os seguintes métodos:
        - `private static boolean verifyExpression(String s)`
            - Retorna true caso a expressão seja válida em relação a "abres” e "fechas”.
        - `private static String showErrorPosition(String s, int position)`

- Retorna uma string com o erro relacionado a "abres” e "fechas”.

- `private static boolean verifySyntax(String s)`
  - Retorna true caso a expressão não tenha erros de sintaxe relacionados ordem corretas de caracteres.
- `private static boolean onlySpaceAllowedArround(String s, int i)`
  - Retorna true caso a expressão tenha um espaço tanto antes quanto depois do caractere da string na posição `i`.
- `static boolean isNumber(char element)`
  - Retorna true caso element seja um numero.
- `static boolean isOperator(char element)`
  - Retorna true caso element seja um operador.
- `static boolean isOpen(char element)`
  - Retorna true caso element seja um "abre”.
- `static boolean isClose(char element)`
  - Retorna true caso element seja um "fecha”.
- `static boolean isDot(char element)`
  - Retorna true caso element seja um ponto.
- `static boolean isSpace(char element)`
  - Retorna true caso element seja um espaço.
- `private static String verifyExpressionError(String s)`
  - Retorna uma string com o erro da expressão relacionado a ordem correta de caracteres.

Resultados

```
-----------------------------------------------------------------------------------------------------------------------------------------------------------

Step 0 -> { ( 5 + 12 ) + [ ( 10 - 8 ) + 2 ] }
Step 1 -> { 17.0 + [ 2.0 + 2 ] }
Step 2 -> { 17.0 + 4.0 }
Step 3 -> 21.0
Expressao: { ( 5 + 12 ) + [ ( 10 - 8 ) + 2 ] };
Resultado: 21.0 ;
Tamanho maximo da pilha: 35;

-----------------------------------------------------------------------------------------------------------------------------------------------------------

Step 0 -> { ( 2 + 3 ) * [ 3 / ( 1 - 3 ) ] }
Step 1 -> { 5.0 * [ 3 / -2.0 ] }
Step 2 -> { 5.0 * -1.5 }
Step 3 -> -7.5
Expressao: { ( 2 + 3 ) * [ 3 / ( 1 - 3 ) ] };
Resultado: -7.5 ;
Tamanho maximo da pilha: 33;

-----------------------------------------------------------------------------------------------------------------------------------------------------------

[[ ERRO ]]
Expressao: { ( 12 + 34 ) * [ ( 47 - 17 / ( 60 - 20 ) ] } ;
Erro: Expressão: { ( 12 + 34 ) * [ ( 47 - 17 / ( 60 - 20 ) ] }
Erro de sintaxe: Está falando um ) na expressão!;

-----------------------------------------------------------------------------------------------------------------------------------------------------------

Step 0 -> { [ ( ( 27 - 18 ) * 3 ) - ( ( 58 + 33 ) - ( ( 108 - 79 ) + 2 ) ) ] + [ ( 5 + 12 ) + ( ( 10 - 8 ) + 2 ) ] }
Step 1 -> { [ ( 9.0 * 3 ) - ( 91.0 - ( 29.0 + 2 ) ) ] + [ 17.0 + ( 2.0 + 2 ) ] }
```

```
Step 2 -> { [ 27.0 - ( 91.0 - 31.0 ) ] + [ 17.0 + 4.0 ] }
Step 3 -> { [ 27.0 - 60.0 ] + 21.0 }
Step 4 -> { -33.0 + 21.0 }
Step 5 -> -12.0
Expressao: { [ ( ( 27 - 18 ) * 3 ) - ( ( 58 + 33 ) - ( ( 108 - 79 ) + 2 ) ) ] + [ ( 5 + 12 ) + ( ( 10 - 8 ) + 2 ) ] };
Resultado: -12.0 ;
Tamanho maximo da pilha: 106;

-----------------------------------------------------------------------------------------------------------------------------------------------------

[[ ERRO ]]
Expressao: { [ [ ( 27 - 18 ) * 3 ] - [ ( 58 + 33 ) - [ ( 108 - 79 ] + 2 ) ] ] + [ ( 5 + 12 ) + ( ( 10 - 8 ) + 2 ) ] } ;
Erro: Expressão: { [ [ ( 27 - 18 ) * 3 ] - [ ( 58 + 33 ) - [ ( 108 - 79 ] + 2 ) ] ] + [ ( 5 + 12 ) + ( ( 10 - 8 ) + 2 ) ] }
Erro de sintaxe: ] no lugar de ) na expressão!;

-----------------------------------------------------------------------------------------------------------------------------------------------------

Step 0 -> { [ ( ( 2 ^ 5 ) - ( 3 * 15 ) ) + ( ( 102 + 379 ) * ( 468 - 248 ) ) ] - [ ( ( 3 ^ 6 ) - ( 54 * 11 ) ) + ( ( 175 / 5 ) / ( 100 - 11
Step 1 -> { [ ( 32.0 - 45.0 ) + ( 481.0 * 220.0 ) ] - [ ( 729.0 - 594.0 ) + ( 35.0 / -17.0 ) ] }
Step 2 -> { [ -13.0 + 105820.0 ] - [ 135.0 + -2.0588235294117645 ] }
Step 3 -> { 105807.0 - 132.94117647058823 }
Step 4 -> 105674.05882352941
Expressao: { [ ( ( 2 ^ 5 ) - ( 3 * 15 ) ) + ( ( 102 + 379 ) * ( 468 - 248 ) ) ] - [ ( ( 3 ^ 6 ) - ( 54 * 11 ) ) + ( ( 175 / 5 ) / ( 100 - 1
Resultado: 105674.05882352941 ;
Tamanho maximo da pilha: 138;

-----------------------------------------------------------------------------------------------------------------------------------------------------

[[ ERRO ]]
Expressao: { [ ( ( 2 ^ 5 ) - ( 3 * 15 ) ) + ( ( 102 + 379 ) * ( 468 - 248 ) ) ] - [ ( ( 3 ^ 6 ) - ( 54 * ) ) + ( ( 175 / 5 ) / ( 100 - 117
Erro: 10 - Expressão deve conter um numero ou inicio de expressão no lugar de um ')' na posição 94.
{ [ ( ( 2 ^ 5 ) - ( 3 * 15 ) ) + ( ( 102 + 379 ) * ( 468 - 248 ) ) ] - [ ( ( 3 ^ 6 ) - ( 54 * >>)<< ) ) + ( ( 175 / 5 ) / ( 100 - 117 ) ) ]

-----------------------------------------------------------------------------------------------------------------------------------------------------

Step 0 -> { [ ( ( ( 4 ^ 4 ) - ( 13 * 15 ) ) + ( ( 123 + 456 ) * ( 987 - 654 ) ) ) + ( ( ( 3 ^ 6 ) - ( 2 * 34 ) ) + ( ( 242 + 353 ) * ( 468
Step 1 -> { [ ( ( 256.0 - 195.0 ) + ( 579.0 * 333.0 ) ) + ( ( 729.0 - 68.0 ) + ( 595.0 * 220.0 ) ) ] - [ ( ( 32.0 - 45.0 ) + ( 481.0 * 220.
Step 2 -> { [ ( 61.0 + 192807.0 ) + ( 661.0 + 130900.0 ) ] - [ ( -13.0 + 105820.0 ) + ( -13.0 + 112554.0 ) ] }
Step 3 -> { [ 192868.0 + 131561.0 ] - [ 105807.0 + 112541.0 ] }
Step 4 -> { 324429.0 - 218348.0 }
Step 5 -> 106081.0
Expressao: { [ ( ( ( 4 ^ 4 ) - ( 13 * 15 ) ) + ( ( 123 + 456 ) * ( 987 - 654 ) ) ) + ( ( ( 3 ^ 6 ) - ( 2 * 34 ) ) + ( ( 242 + 353 ) * ( 468
Resultado: 106081.0 ;
Tamanho maximo da pilha: 284;

-----------------------------------------------------------------------------------------------------------------------------------------------------

[[ ERRO ]]
Expressao: { [ ( ( ( 4 ^ 4 ) - ( 13 * 15 ) ) + ( ( 123 + 456 ) * ( 987 - 654 ) ) ) + ( ( ( 3 ^ 6 ) - ( 2 * 34 ) ) + ( ( 242 + 353 ) * ( 468
Erro: Expressão: { [ ( ( ( 4 ^ 4 ) - ( 13 * 15 ) ) + ( ( 123 + 456 ) * ( 987 - 654 ) ) ) + ( ( ( 3 ^ 6 ) - ( 2 * 34 ) ) + ( ( 242 + 353 ) *
Erro de sintaxe: Está falando um ) na expressão!;
```

Conclusão

Durante o desenvolvimento, obtivemos maiores dificuldades na confecção da classe `ExpressionSimplifierService`, por conta da lógica por trás de sua execução, lidando diretamente com regras de negócio da aplicação e operações aritméticas. O fato da solução estar sendo dividida em camadas de abstração não a caracteriza como uma solução complexa. A mesma é simples e, em sua essência, consiste em substituir as expressões dentro dos "abre" e "fecha", substituí-las pelo seu resultado e, em seguida, reprocessa-la até que uma solução seja atingida. Na nossa aplicação temos a complexidade O(n), pois não temos nenhum laço encadeado na solução de uma expressão individual. Os resultados foram comparados com cálculos realizados na plataforma Ocatve online e formam considerados válidos pelos integrantes do gurpo.