Instituto Federal do Rio Grande do Sul
Campus Porto Alegre

Academia

Disciplina: Verificação e Validação de Sistemas

Professor:
Rodrigo Prestes Machado

Aluna:
Karina Barbosa da Rosa

Sumário

1. Introdução

Este documento fornece informações dos diversos tipos de testes feitos na disciplina de VVS. Assim como as ferramentas utilizadas. Os projetos encontram-se disponíveis em:
https://github.com/Kahrosa/testesunitarios.git,
https://github.com/Kahrosa/testesintegrados.git,
https://github.com/Kahrosa/testeselenium.git

2.Testes estáticos com PMD

Este projeto visa criar um ambiente adequado para o desenvolvimento de diversas formas de teste em Java.

A criação do teste começa em várias etapas, entre as quais podemos citar o processo de verificação do código através do PMD, que é responsável por verificar estaticamente o código, ou seja, analisar o programa sem executar o programa. No entanto, quando consideramos as inspeções no código-fonte, atualmente temos um grande número de ferramentas capazes de realizar esse tipo específico de análise.

PMD é uma ferramenta para analisar código-fonte escrito em Java. Ele possui um grande número de regras de análise e pode investigar desde estilos de código até questões mais complexas, como segurança e desempenho. Você também pode criar novas regras no PMD, ou seja, é uma ferramenta flexível que permite modificar seu uso em diferentes tipos de projetos Java.

Podemos usar o PMD nas fases de desenvolvimento, construção e instalação do sistema.

Além de usar estaticamente o PMD no código Java, você também pode se conectar ao Maven, que é um plug-in para verificar o código Java.

Portanto, o PDM e o plug-in Maven podem incorporar verificações estáticas no processo de integração contínua. Portanto, mesmo antes de o código ser compilado, podemos realizar análises e usar parâmetros de qualidade para decidir se devemos continuar a integrar novos fragmentos de código (funções, correções de defeitos, etc.) no sistema.

Teste realizado no terminal do VSCode

O teste com PMD foi realizado com um teste unitário onde a regra aplicada foi esta:

CommentRequired

Desde: PMD 5.1

Prioridade: Média (3)

Indica se os comentários javadoc (formais) são necessários (ou indesejados) para elementos específicos da linguagem.

Esta regra é definida pela seguinte classe Java: net.sourceforge.pmd.lang.java.rule.documentation.CommentRequiredRule

```
Código: <rule ref="category/java/documentation.xml/CommentRequired" />
```

Foram aplicados os comandos: pmd, mvn clean pmd:pmd package e mvn pmd:pmd rodando com sucesso.

3. Testes Unitários com Junit

O teste de unidade estabelece o processo de teste de pequenos componentes métodos ou classes do programa. Portanto, esse tipo de teste envolve a chamada de rotinas com diferentes parâmetros de entrada para realizar todas as ações de um trecho de código.

O Junit é a principal ferramenta para teste de unidade na linguagem Java.
Exemplo aplicado no teste:

No exemplo que desenvolvi mediante as tarefas a nós destinadas, a anotação @Test indica que addition (adição) é um método de teste. Por sua vez, a assertiva assertEquals verifica se o resultado da soma de 1+1 por meio do método add da classe Calculator retorna no valor 2. Em sequência realizei a subtração do mesmo teste unitário de nome Calculator, retornando o resultado zero, pois 1-1 é zero.

Os próximos testes de unidade são contemplados com anotações do Junit que ajudam a configurar o teste, nessas anotações são incluidas: @ BeforeAll, @ AfterAll, @ BeforeEach e @AfterEach.
@BeforeAll: indica que o método estático será executado antes dos outros métodos.
@AfterAll: indica que métodos estáticos serão executados após outros métodos.
@BeforeEach: Indica o método que será executado antes que cada método seja anotado com o seguinte: @ Test, @ RepeatedTest, @ ParameterizedTest ou @TestFactory.
@AfterEach: indica o método que será executado após cada método ser anotado com o seguinte: @ Test, @ RepeatedTest, @ ParameterizedTest ou @TestFactory.

O exemplo aplicado no projeto foi usado a anotação @BeforeAll. No exemplo, o método init estático será executado apenas uma vez antes de executar qualquer teste. Por outro lado, o método add possui a anotação @BeforeEach e será executado antes de cada método anotado com @Test, ou seja, o exemplo a seguir

fará com que add execute duas vezes. A anotação @DisplayName, permite adicionar um nome mais significativo ao teste. A ordenação de um teste também é importante, portanto para nos ajudar temos a anotação @Order que estabelece uma sequencia pré-definida. Na sequencia temos a presença da anotação @Order, que faz com que o segundo método de teste (second) seja executado antes do primeiro.
Correlacionado com o Junit,

O exemplo incorporado na sequência desenvolvido com as tarefas da disciplina foi AnnotationsTest, onde para poder usar o Junit como teste por meio do Surefire, ele deve estar em conformidade com o padrão de nome estabelecido pelo plug-in, por exemplo, usar o sufixo Teste para nomear todas as classes Java que implementam o teste. Portanto, uma vez incorporado ao projeto Maven, foi testado durante o ciclo de teste com o seguinte comando: mvn test.

O agrupamento do teste unitario é feito através da tag: @tag. Esta é utilizada para que possam ser executados separadamente de acordo com os requisitos.

4. Teste de integração com o Failsafe

Failsafe é um dos plug-ins Maven para a execução de testes de integração. A principal diferença entre Failsafe e Surefire é que no primeiro teste, se o teste falhar, o processo de construção do sistema não será afetado.

O plug-in à prova de falhas tem duas finalidades:

Failsafe: Teste de integração - execute o teste de integração do aplicativo

Failsafe: verificação - verifique se o teste de integração do aplicativo foi aprovado

No exemplo do projeto o Failsave estava configurado no projeto Maven, e só foi necessário executar o comando verify para verificar o Failsave: mvn verify

O teste de integração no projeto Academia foi criado com as classes DAO e Repository, as quais são nesta mesma ordem (DAO – Data Access Object), utilizado para persistência de dados e a classe Repository implementa todo o crud, no caso, deletar e atualizar.

Está também presente no projeto Academia o MVC (Model, View e Controller) onde o Model tem a responsabilidade de apresentar a classe Usuário onde encontra-se o id e o nome do usuário.

O view é responsável pela interface que será apresentada futuramente, onde um botão dispara um método no controller que acessa a classe DAO (banco) que realiza uma ação com os dados presentes.

O controller é exatamente o crud, onde realiza a função de recuperar, salvar, deletar e atualizar.

5. Selenium IDE, Side Runner e integração com o Junit/Manven

Selenium é uma ferramenta que cria e copia testes funcionais. As principais ferramentas incluem: Selenium IDE e Selenium Side Runner.

O IDE é um plug-in para Chrome ou Firefox, que pode gravar, editar e depurar testes funcionais na web. Outra característica do IDE é que ele pode exportar testes na linguagem Java / Junit, melhorando o processo de teste e também integrar-se com o processo contínuo Maven. Por outro lado, o Side Runner é um software baseado em Nodejs, pronto para executar os testes anteriormente exportados pelo Selenium IDE através do interpretador de comandos.

No exemplo desenvolvido para o projeto, utilizou-se o Selenium IDE para o FireFox onde foi gerado o arquivo ifrs.side e posteriormente exportado para depois ser utilizado no exemplo do projeto.

A partir disto foi instalado o Selenium Side Runner que é um aplicativo que permite iniciar um navegador a partir da linha de comando para executar casos de teste no formato .side.

Então o teste aplicado executa as seguintes etapas:

(1) executar HTTP GET a partir da URL (ifrs.edu.br),

(2) definir o tamanho da janela do navegador (opcional),

(3) navegar no link Editais,

(4) localizar CSS Elementos da classe .editais__title, e

(5) Se um elemento contendo o CSS requerido for encontrado, o teste é aprovado.

6. Tabelas

Teste PMD	
Nome	Regras
Objetivo	Fazer com que todo código esteja comentado.

Teste de Unitário	
Nome	CalculatorTest
Objetivo	Testar a soma e a subtração do teste implementado.

Teste de Unitário	
Nome	AnnotationsTest
Objetivo	Testar de modo ordenados com a tag @Order, melhorar especificações do texto com a tag @tag Adicionar e remover informações do teste.

Teste de Integração	
Nome	Academia
Objetivo	Fazer integração entre os 2 containers juntamente com a classe que irá implementar o teste. Cadastrando o ide e o nome do usuário.

Teste de Selenium	
Nome	Edital
Objetivo	(1) executar HTTP GET a partir da URL (ifrs.edu.br), (2) definir o tamanho da janela do navegador (opcional), (3) navegar no link Editais, (4) localizar CSS Elementos da classe .editais__title, e (5) Se um elemento contendo o CSS requerido for encontrado, o teste é aprovado.