UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
PRÓ-REITORIA DE GRADUAÇÃO
DEPARTAMENTO DETEC
CURSO BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO

JHOAN FERNANDES DE OLIVEIRA
DAYVISON EYC DE MOURA SILVA
CLAUDIO CAUEH OLIVEIRA XAVIER

TÍTULO: BOOKSWAP

PAU DOS FERROS
2024

TÍTULO: BOOK SWAP

Relatório de testes

Professor: Alysson Filgueira Milanez, Prof. Dr.

PAU DOS FERROS

2024

RESUMO

O “BookSwap” é um aplicativo que muda a maneira como os leitores interagem com seus livros e uns com os outros, ele permite que os usuários cadastrem seus próprios livros e os troquem com outros, criando uma comunidade de leitores engajados, este aplicativo não só incentiva a leitura, mas também promove a sustentabilidade ao reutilizar livros e proporciona uma maneira econômica de explorar novos títulos e autores.

Palavras-chave: BookSwap, leitura, educação, sustentabilidade, cadastro de livros, troca de livros, comunidade de leitores, descoberta de novos títulos e autores, gerenciamento de livros, comunicação entre usuários.

1 INTRODUÇÃO

O “BookSwap” é um aplicativo inovador que transforma a interação dos leitores com seus livros e entre si, criando uma comunidade de leitores engajados. Ele promove a leitura, a educação e a sustentabilidade, permitindo que os usuários cadastrem seus próprios livros para troca, descubram novos títulos e autores, e se conectem com outros leitores.

O aplicativo possui funcionalidades essenciais que permitem o cadastro e gerenciamento de livros, incluindo detalhes como título, autor e disponibilidade. Ele também facilita a busca por livros disponíveis para troca e permite a comunicação entre os usuários para acordar as trocas. Desta forma, o BookSwap não é apenas uma plataforma de troca de livros, mas também uma comunidade onde os leitores podem compartilhar suas paixões literárias.

2 TESTES

2.1 TESTE DE LOGIN

A classe Login é responsável por gerenciar o processo de login no aplicativo. Ela contém um método principal também chamado login, que é usado para autenticar um usuário. Aqui está um resumo do funcionamento do método:

1. **Inicialização**: O método começa inicializando duas variáveis, **loggedIn** e tentativas. **loggedIn** é um booleano que indica se o usuário está logado ou não, e tentativas é um contador para o número de tentativas de login.
2. **Loop de Login**: O método entra em um loop while que continua até que o usuário esteja logado ou o número máximo de tentativas seja atingido.
3. **Solicitação de Dados do Usuário**: Dentro do loop, o método solicita ao usuário que insira seu login e senha. Ele faz isso usando o Scanner passado como parâmetro para ler as entradas do usuário.
4. **Verificação do Usuário**: Em seguida, o método tenta obter o usuário correspondente ao login e senha fornecido através do método UsuariosRepo.getInstance().getUsuario(login, senha).
5. **Verificação de Sucesso do Login**: Se o usuário for encontrado, isso significa que o login foi bem-sucedido. O método então imprime uma mensagem de sucesso, define **loggedIn** como *true* e chama o método *Menu.menu(scanner, user)* para levar o usuário ao menu principal do aplicativo.

6. **Tratamento de Falha no Login**: Se o usuário não for encontrado, isso significa que o login falhou. O método então lança uma exceção com uma mensagem de erro.
7. **Tratamento de Exceções**: O método login (Scanner scanner) tem um bloco catch que captura qualquer exceção lançada durante o processo de login. Quando uma exceção é capturada, o método imprime uma mensagem de erro e pede ao usuário que tente novamente. Ele também incrementa o contador tentativas.
8. **Opção de Sair**: Após uma falha no login, o método dá ao usuário a opção de continuar tentando fazer login ou sair. Se o usuário escolher sair, o método retorna e o processo de login termina.
9. **Limite de Tentativas**: Se o número máximo de tentativas for atingido e o usuário ainda não estiver logado, o método imprime uma mensagem informando que o limite de tentativas foi excedido e o cadastro foi cancelado. Em seguida, ele retorna, encerrando o processo de login.

Resumo dos problemas que encontrados e como os resolvemos:

1. **Erro "java.util.NoSuchElementException: No line found"**: Este erro ocorreu porque o método Scanner.nextLine() foi chamado, mas não havia mais linhas de entrada disponíveis. Para resolver isso, modificamos o código para verificar se o Scanner tem uma próxima linha antes de tentar lê-la, usando o método Scanner.hasNextLine().
2. **Erro "java.lang.StackOverflowError"**: Este erro ocorreu porque o método Login.login(Scanner scanner) estava chamando a si mesmo recursivamente, resultando em um StackOverflowError. Para resolver isso, substituímos a chamada recursiva por um loop.
3. **Teste infinito**: Este problema ocorreu porque o método Menu.menu(scanner, user) estava em um loop infinito. Para resolver isso, adicionamos um parâmetro booleano isTesting ao método e modificamos o código para sair do loop quando isTesting é true.
4. **Erro "Expected java.lang.Exception to be thrown, but nothing was thrown"**: Este erro ocorreu porque o teste testLoginFail() esperava que uma exceção fosse lançada quando um login falha, mas uma exceção não foi lançada. Para resolver isso, modificamos o teste para verificar se a mensagem de erro esperada é impressa, em vez de esperar que uma exceção seja lançada.

5. **Problemas com instituição pré-cadastrado**: Após alguns testes realizados em Instituição foram feitas alterações no código, pois o método de cadastro de instituição agora retorna uma mensagem caso haja alguma outra já cadastrada com o mesmo nome. Então, agora o **setUp** do **LoginTest** faz essa verificação antes de adicionar uma nova instituição para testes.
6. **A seguir algumas imagens do processo**:

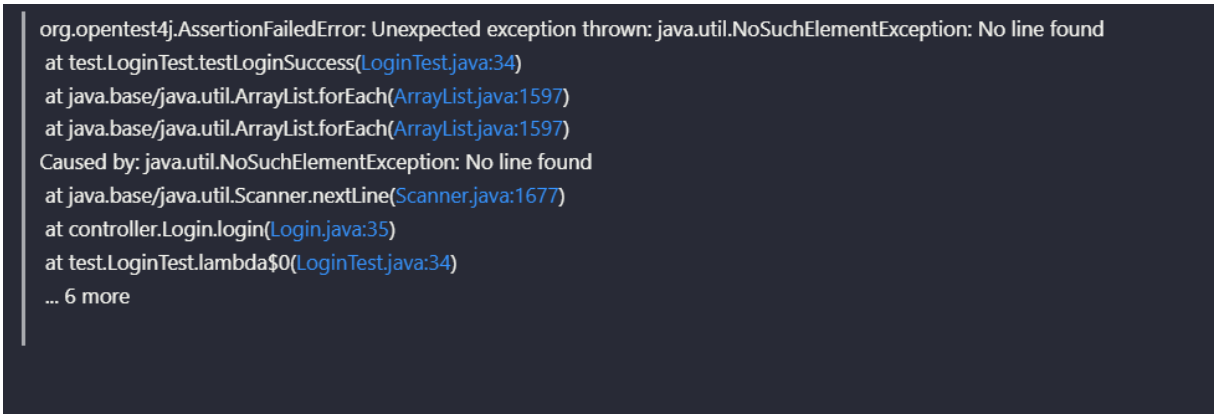

Figura 1: Erro de Buffer

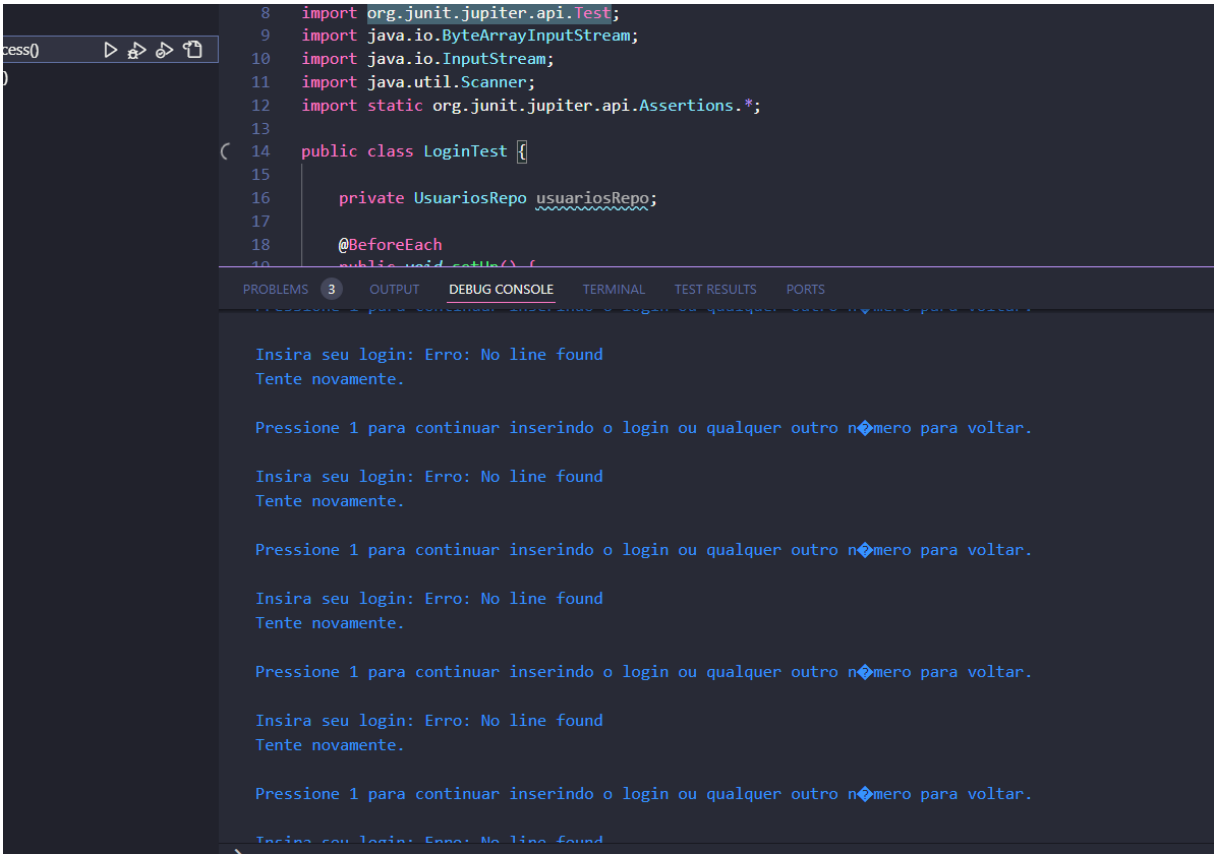

Figura 2: Muitas tentativas atingidas

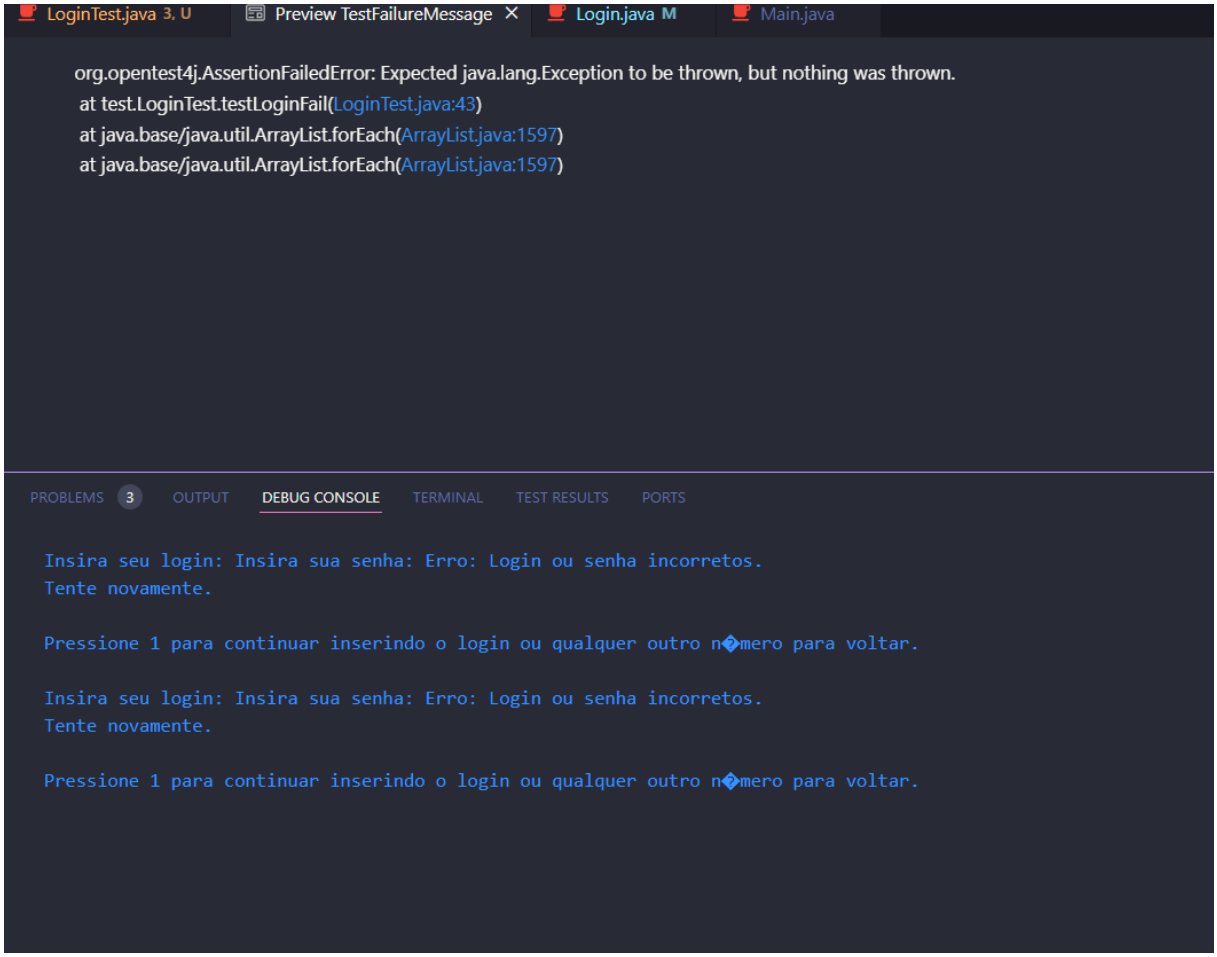

Figura 3: Java Lang Exception

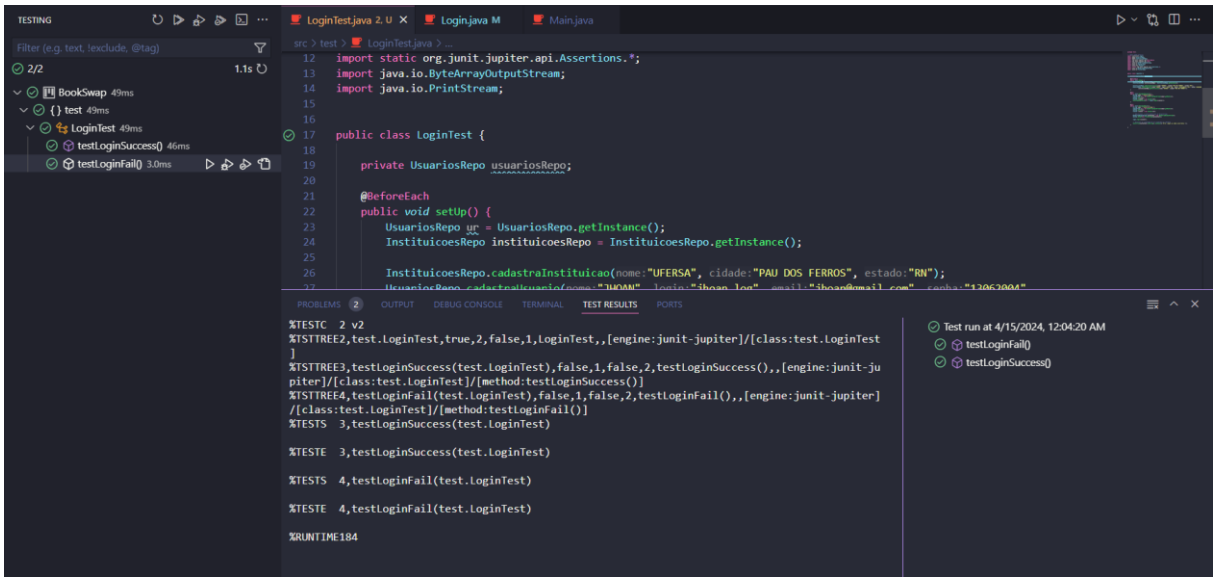

Figura 4: Teste bem sucedido

Por fim, o login foi testado com sucesso para as situações propostas – um caso com um login cadastrado resultando em sucesso e um com um login não cadastrado resultando em fail –. A funcionalidade atendeu a necessidade, mas entendemos que para ter mais confiança acerca do método seriam necessários mais testes e mais rebuscados. Além disso, a solução proposta para evitar que o usuário fique infinitamente digitando login foi a mais simples possível (após algumas poucas tentativas o programa fecha). O grupo entende que seria ideal deixar o usuário em espera por algumas horas até que ele pudesse tentar o login novamente, dessa forma evitando ataques por estresse. Contudo, devido a questões de tempo e prioridades optamos por apenas realizar a ação citada anteriormente.

2.2 TESTE DE CADASTRO

A classe Cadastro é como o porteiro do nosso aplicativo de troca de livros. Ela verifica as credenciais dos usuários e só permite a entrada daqueles que estão devidamente cadastrados.

Agora, imagine que você é um detetive tentando garantir que esse porteiro esteja fazendo seu trabalho corretamente. É aí que entram os testes. No caso da classe Cadastro, criamos um teste para verificar se o método de cadastro está funcionando como esperado.

No teste de sucesso do cadastro, fornecemos dados válidos e esperamos que o cadastro seja bem-sucedido. É como se estivéssemos testando se o porteiro permite a entrada de um visitante com um crachá válido.

No teste de falha do cadastro, fornecemos dados inválidos e esperamos que o cadastro falhe. É como se estivéssemos testando se o porteiro impede a entrada de um visitante sem crachá ou com um crachá inválido.

Ao realizar esses testes, podemos ter certeza de que nosso porteiro, a classe Cadastro, está fazendo seu trabalho corretamente, permitindo apenas usuários válidos e mantendo fora os inválidos. E assim, mantemos nosso aplicativo seguro e funcionando sem problemas.

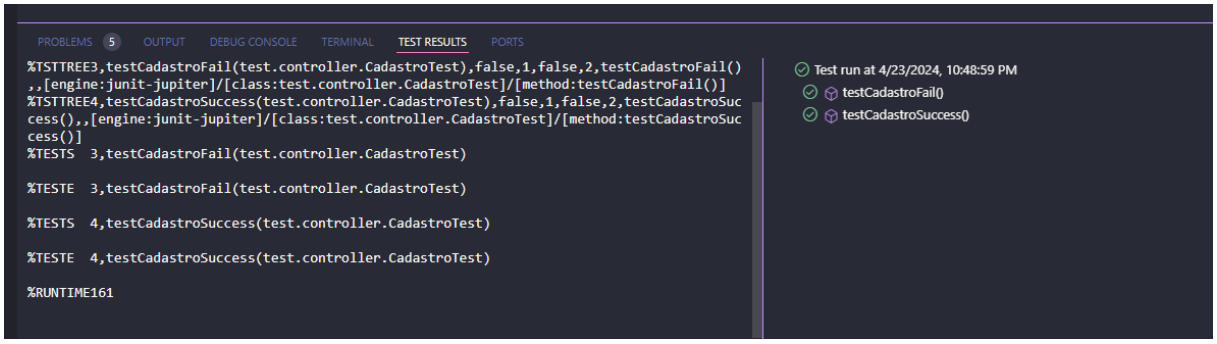

Figura 5: Teste passou

2.3 TESTE DO REPOSITÓRIO DE USUÁRIOS

No decorrer do nosso trabalho, realizamos uma série de testes na classe **UsuariosRepo** do projeto de troca de livros entre alunos de universidade. Esta classe é responsável por gerenciar os usuários do sistema, incluindo a criação, armazenamento e recuperação de usuários.

A classe **UsuariosRepo** é uma classe *singleton*, o que significa que apenas uma instância dessa classe pode existir durante a execução do programa. Ela contém uma lista de usuários e fornece métodos para adicionar usuários à lista, recuperar todos os usuários, recuperar um usuário pelo nome ou pelo nome e senha, verificar a existência de um login e obter a quantidade de usuários. Além disso, a classe também fornece métodos estáticos para cadastrar um usuário e verificar um login.

Para garantir que a classe **UsuariosRepo** funcione corretamente, criamos uma classe de teste separada usando a estrutura de teste unitário JUnit. Nesta classe de teste, criamos vários testes para cobrir todos os métodos na classe **UsuariosRepo**. Cada teste cria uma nova instância da classe **UsuariosRepo**, adiciona alguns usuários e verifica se os métodos da classe estão retornando os resultados esperados.

Durante os testes, encontramos um problema com o método **addUsuario**. O teste para este método estava falhando porque o número de usuários na lista era maior do que o esperado. Após uma investigação mais aprofundada, descobrimos que o problema era que os testes anteriores estavam adicionando usuários à lista e esses usuários não estavam sendo removidos após cada teste. Para resolver este problema, adicionamos um método **tearDown** que é chamado após cada teste para limpar a lista de usuários.

Com esses testes, conseguimos garantir que a classe **UsuariosRepo** está funcionando corretamente e que todos os seus métodos estão retornando os resultados esperados. Isso nos

dá confiança de que a classe será capaz de gerenciar os usuários do sistema de maneira eficaz e eficiente. Continuaremos a monitorar e testar a classe **UsuariosRepo** conforme avançamos no desenvolvimento do projeto para garantir que ela continue a atender às nossas necessidades.

Figura 6: Erro no AssertEquals

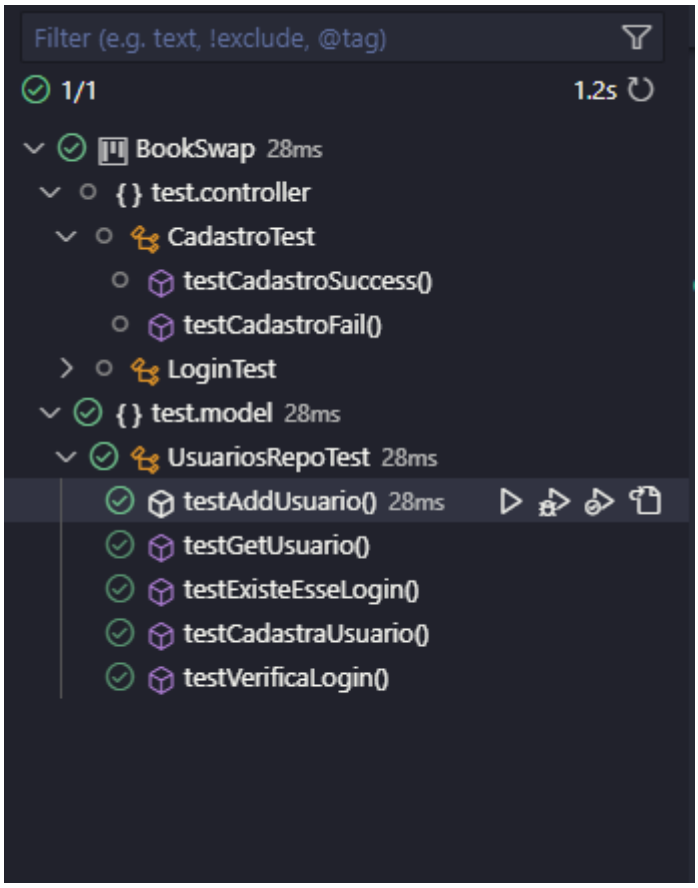

Figura 7: Teste passou

2.4 TESTE DO REPOSITÓRIO DE BIBLIOTECA

Esta classe é responsável por gerenciar a biblioteca do sistema, incluindo a criação, armazenamento e recuperação de livros.

A classe **BibliotecaRepo** é uma classe *singleton*, o que significa que apenas uma instância dessa classe pode existir durante a execução do programa. Ela contém uma biblioteca e fornece métodos para obter a biblioteca e remover a biblioteca.

Para garantir que a classe **BibliotecaRepo** funcione corretamente, criamos uma classe de teste separada usando a estrutura de teste unitário JUnit. Nesta classe de teste, criamos vários testes para cobrir todos os métodos na classe **BibliotecaRepo**. Cada teste cria uma nova instância da classe **BibliotecaRepo**, adiciona alguns livros à biblioteca e verifica se os métodos da classe estão retornando os resultados esperados.

Durante os testes, encontramos um problema com os métodos **testAdicionarERemoverLivro** e **testRemoverLivroDeBibliotecaVazia**. Os testes estavam falhando porque a biblioteca era null quando tentávamos adicionar ou remover um livro dela. Isso foi causado pelo método **removerBiblioteca** na classe **BibliotecaRepo**, que define a biblioteca como null. No entanto, o método **getBiblioteca** não verificava se a biblioteca era null antes de retorná-la. Resolvemos este problema modificando o método **getBiblioteca** para criar uma nova biblioteca se a atual fosse *null*.

Com esses testes, conseguimos garantir que a classe **BibliotecaRepo** está funcionando corretamente – dentro dos poucos testes aplicados – e que todos os seus métodos estão retornando os resultados esperados. Isso nos dá confiança de que a classe será capaz de gerenciar a biblioteca do sistema de maneira eficaz e eficiente para o momento. Se o projeto continuar, muito provavelmente serão necessários mais testes para as funções de biblioteca, inclusive testes de integração.

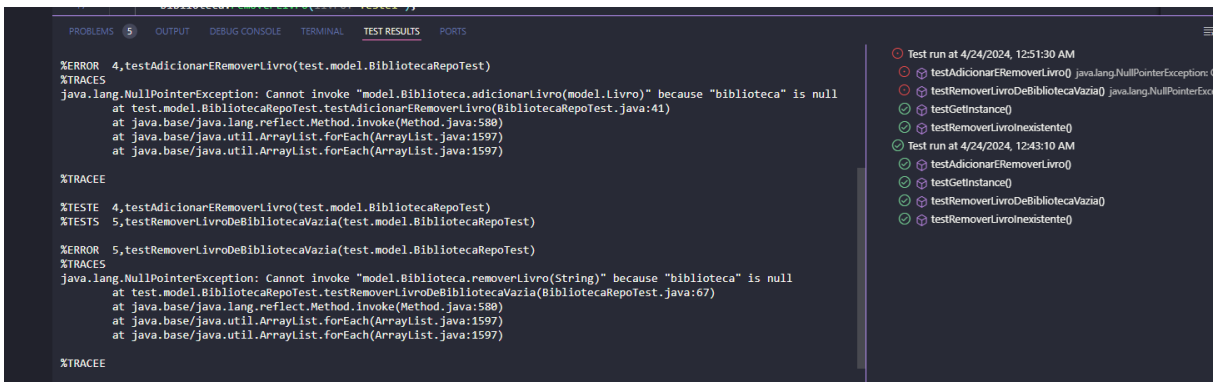

Figura 8: Problemas com remover biblioteca vazia

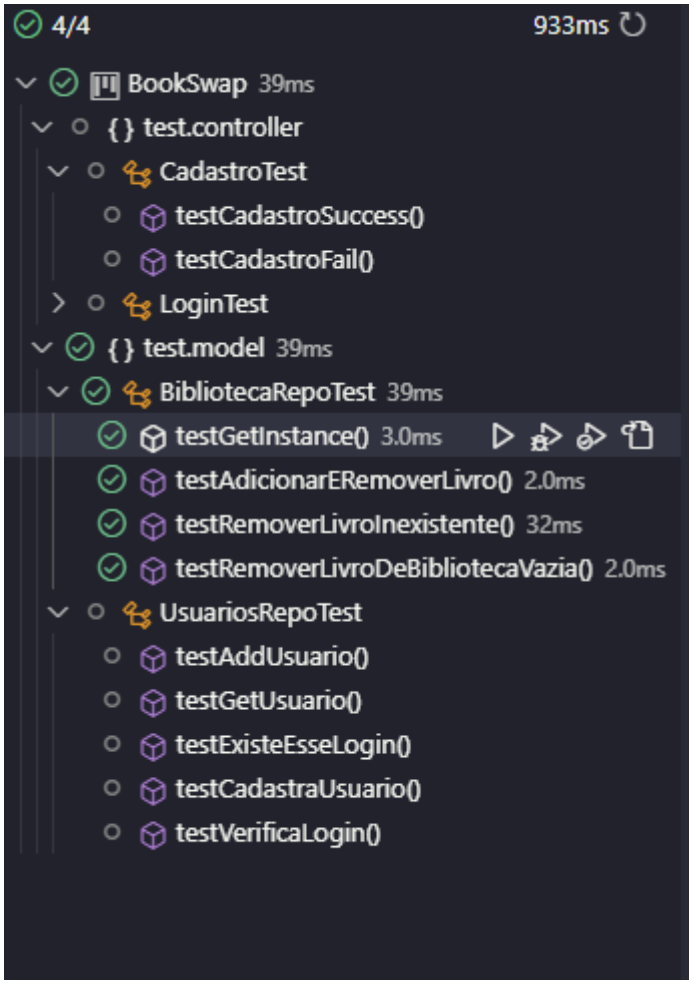

Figura 9: Teste passou

2.5 TESTE DE TROCA

A classe **Troca** é uma parte crucial do nosso projeto. Ela é a responsável por gerenciar as trocas de livros entre os usuários. Cada troca envolve dois livros, que são trocados entre dois usuários. A classe também mantém o controle do estado da troca, que pode ser aprovada, rejeitada ou pendente.

Para garantir que ela esteja funcionando corretamente, criamos uma série de testes unitários usando o *JUnit*. Esses testes verificam os principais aspectos da classe, desde a criação de uma nova troca até a aprovação e rejeição de uma troca.

Durante os testes, encontramos um problema com a lógica de aprovação e rejeição. Descobrimos que uma troca poderia ser aprovada mesmo depois de ter sido rejeitada, e vice-versa. Isso foi evidenciado pelo teste **testRejeitarEAprovar**, que estava retornando um resultado inesperado. Para resolver esse problema, modificamos os métodos aprovar e rejeitar

na classe Troca para garantir que, uma vez que uma troca seja aprovada, ela não possa ser rejeitada, e vice-versa.

Embora todos os nossos testes tenham passado, entendemos que isso não garante a perfeição das funções. Sabemos que, em programação, é quase impossível prever todas as possíveis interações do usuário com o sistema. Portanto, mesmo com os resultados positivos dos nossos testes, estamos cientes de que sempre há espaço para melhorias e ajustes. Portanto, com os testes criados esperamos reduzir a taxa de erros o máximo possível.

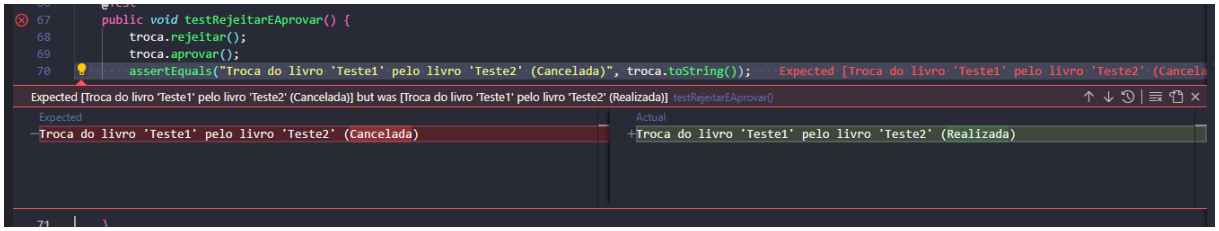

Figura 10: Retorno não esperado em Rejeitar Troca

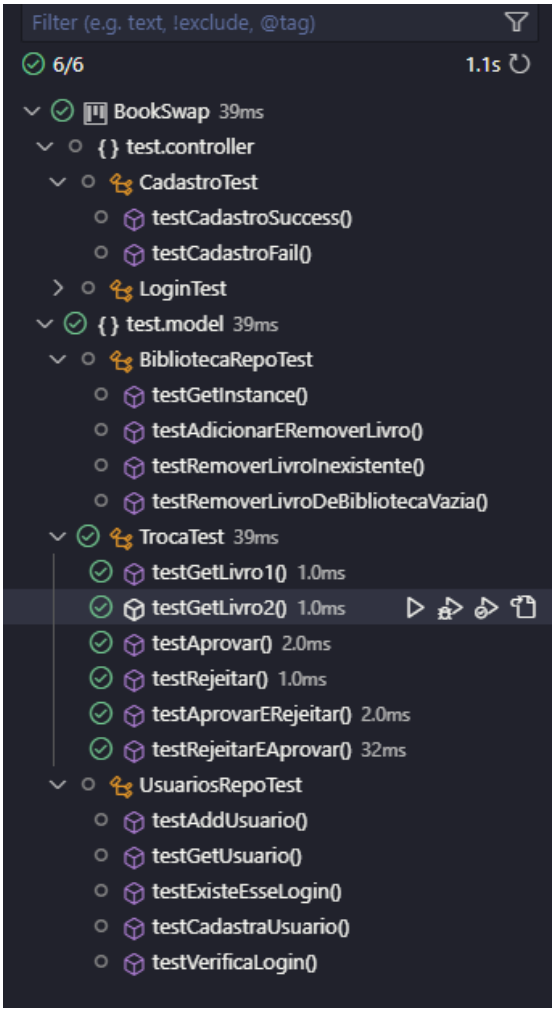

Figura 11: Teste passou

2.6 TESTE DE TRATATIVAS

A classe Tratativas desempenha um papel fundamental no projeto, fornecendo uma série de métodos utilitários para validar e processar dados. Para garantir que esses métodos funcionem corretamente, realizamos uma série de testes unitários.

Os testes abordaram uma variedade de cenários, incluindo entradas válidas e inválidas. Por exemplo, testamos o método **isValidoLogin** com logins que atendiam aos critérios (letras, números, sublinhados ou pontos e com pelo menos três caracteres) e com logins que não atendiam. Da mesma forma, testamos os métodos **isValidoEmail**, **contemDigitos** e **contemDoisEspacosSeguidos** com uma variedade de entradas para garantir que eles se comportassem como esperado.

No entanto, encontramos um desafio ao testar o método **leEVerificaLogin**. Este método estava esperando uma entrada do usuário, o que não é possível em um teste unitário. Para resolver isso, modificamos o método para aceitar uma *String* como parâmetro em vez de um Scanner. Isso permitiu que nós fornecêssemos a entrada diretamente durante os testes.

Além disso, para acomodar essa mudança, também foi necessário modificar a classe Cadastro para se adequar ao novo **leEVerificaLogin**. Com essas modificações, fomos capazes de testar completamente o método e garantir que ele funcione corretamente.

Além disso, outros dois métodos (**testLeEVerificaCidadeInstituicao** e **testLeEVerificaNomeInstituicao**) enfrentaram o mesmo problema que o método de verificação de login. Contudo, nesse caso optamos por não os modificar da mesma forma como aconteceu com o de login. A escolha para isso é que traria novos problemas para refatorar a classe de cadastro e refazer a lógica. Dessa forma, optamos por testes manuais para esses dois.

Por fim, todos os outros métodos finalmente passaram nos testes para as situações em que foram submetidos, garantindo mais confiança em relação a eles separadamente. Não testamos entradas avançadas de teclado ou formas mirabolantes de quebrar o login, como por exemplo entradas em outros idiomas.

Figura 12: Problemas com o Scanner

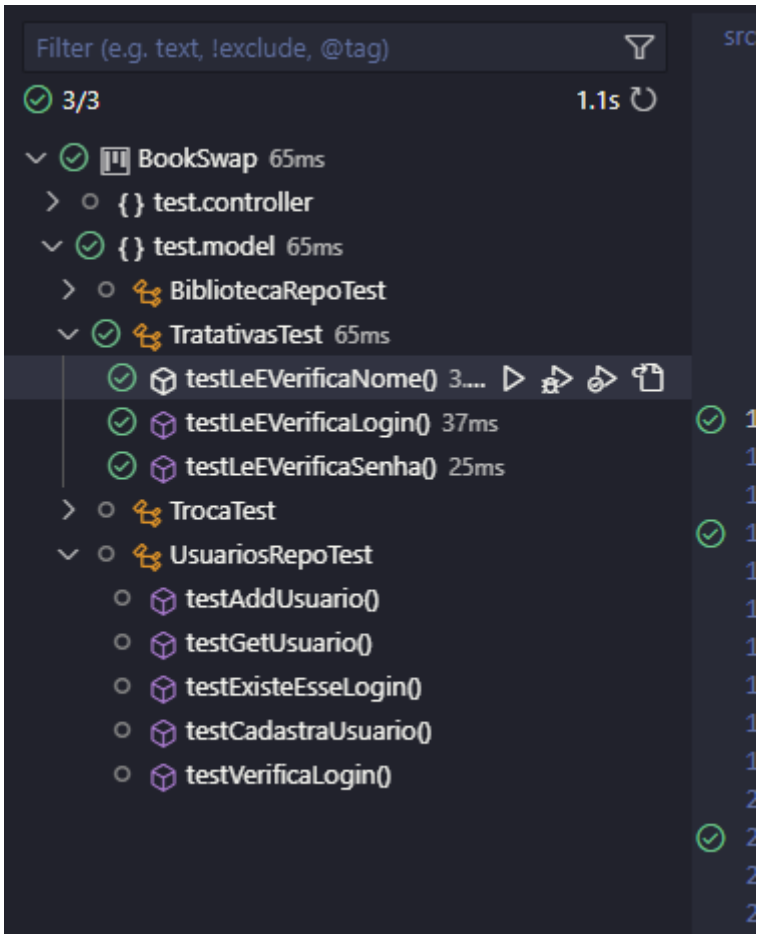

Figura 13: Outros testes passaram

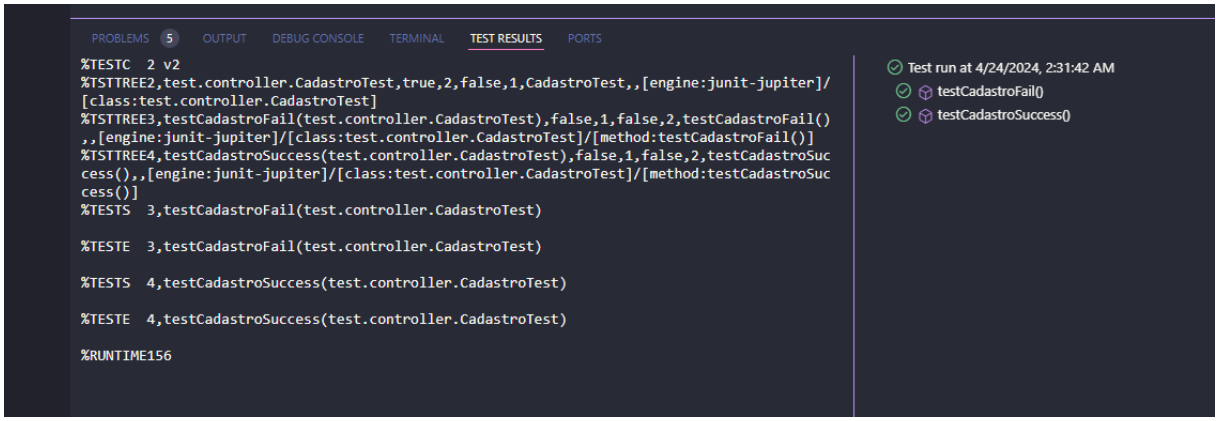

Figura 14: Reteste de Cadastro

Figura 15: IsValidLogin não tratava null

2.7 TESTE DO REPOSITÓRIO DE INSTITUIÇÕES

A classe **InstituicoesRepo** desempenha um papel crucial no projeto, atuando como o repositório para todas as instituições. Ela mantém uma lista de instituições e fornece métodos para adicionar instituições à lista, recuperar todas as instituições, recuperar uma instituição específica pelo nome e pela cidade, verificar a existência de uma instituição pelo nome ou cidade, e obter a quantidade de instituições. Além disso, a classe implementa o padrão *Singleton* para garantir que apenas uma instância dessa classe seja criada.

Para garantir que funcione corretamente, realizamos uma série de testes unitários. Os testes foram projetados para cobrir uma variedade de cenários e garantir que cada método na classe **InstituicoesRepo** funcione como esperado.

Testamos o método **addInstituicao** para garantir que ele adicione corretamente uma instituição à lista e atualize a quantidade de instituições. Também verificamos se ele lança uma exceção quando tentamos adicionar uma instituição com um nome ou cidade vazios, ou uma instituição duplicada.

O **java.lang.IllegalArgumentException** é uma exceção não verificada que é comumente usada para sinalizar que um método recebeu um argumento ilegal ou inadequado.

É uma prática recomendada lançar **IllegalArgumentException** em um método para indicar que um argumento de entrada não atende a um certo critério.

No contexto do método **cadastraInstituicao**, encontramos situações em que era necessário lançar um **IllegalArgumentException**. Especificamente, queríamos garantir que o método não aceitasse uma instituição com um nome ou cidade vazios, ou uma instituição duplicada.

Para lidar com instituições com um nome ou cidade vazios e com instituições duplicadas, adicionamos verificações no início do método **cadastraInstituicao** para lançar um **IllegalArgumentException** se o nome ou a cidade fossem nulos ou vazios. Aqui está o código correspondente:

```
public static void cadastraInstituicao(String nome, String cidade, String estado) {
    InstituicoesRepo listInst = InstituicoesRepo.getInstance();
    if (nome == null || nome.isEmpty()) {
        throw new IllegalArgumentException(s:"Nome não pode ser vazio");
    } else if (cidade == null || cidade.isEmpty()) {
        throw new IllegalArgumentException(s:"Cidade não pode ser vazia");
    } else if (listInst.existeEssaInstituicaoN(nome) && listInst.existeEssaInstituicaoC(cidade)) {
        throw new IllegalArgumentException(s:"Instituição já cadastrada");
    }
    int id = listInst.getQntInstituicoes() + 1000;
    listInst.addInstituicao(new Instituicao(id, nome, cidade, estado));
}
```

Figura 16: Método cadastraInstituicao

Os métodos **getInstituicao**, **existeEssaInstituicaoN** e **existeEssaInstituicaoC** foram testados para garantir que eles retornem as informações corretas. Verificamos se eles retornam à instituição correta ou indicam corretamente a existência de uma instituição quando a instituição existe. Também verificamos se eles retornam *null* ou indicam que a instituição não existe quando a instituição não está na lista.

Todos os testes passaram, o que indica que a classe **InstituicoesRepo** está funcionando corretamente para os testes aplicados e em caráter de método. No entanto, é importante notar que os testes unitários são apenas uma parte do processo de garantia de qualidade e devem ser complementados com outros tipos de testes, como testes de integração e testes de sistema, para garantir a robustez e a confiabilidade do software.

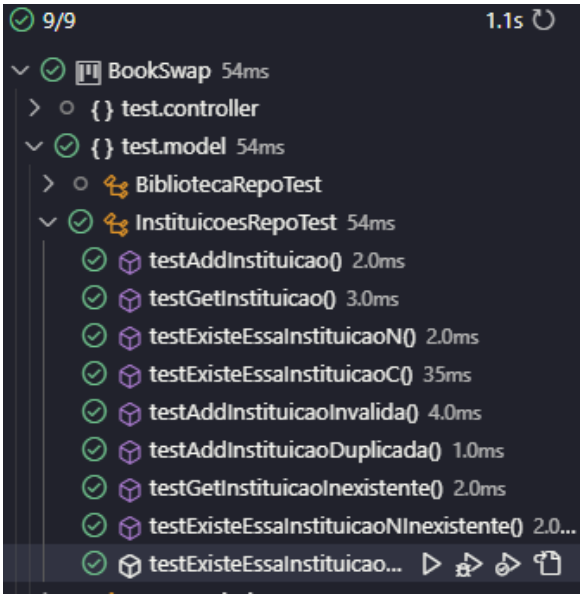

Figura 17: Teste passou

3 CONSIDERAÇÕES FINAIS

Ao longo do desenvolvimento do programa BookSwap, a implementação de testes desempenhou um papel crucial para garantir a qualidade e a robustez do software. Os testes permitiram identificar e corrigir problemas em tempo hábil, além de garantir que as funcionalidades do sistema estivessem funcionando conforme o esperado. Apesar de bugs ainda poderem e com certeza irão existir.

Os testes abordaram aspectos críticos do sistema, como o processo de login, cadastro de usuários, troca de livros, cadastro de instituições e troca de livros. Eles ajudaram a garantir que apenas usuários com credenciais válidas pudessem acessar o sistema e que novos usuários pudessem ser registrados corretamente. Durante os testes, foram identificados e corrigidos vários problemas, incluindo exceções inesperadas e loops infinitos.

O desenvolvimento do programa em si foi um processo iterativo e incremental, com ênfase na escrita de código limpo e manutenível. O padrão de arquitetura Model-View-Controller (MVC) foi adotado para separar a lógica de negócios, a interface do usuário e o controle do fluxo de dados. Isso resultou em um código mais organizado e fácil de entender.

As funcionalidades do programa foram projetadas para atender às necessidades dos usuários de uma maneira eficiente e intuitiva. O programa permite que os usuários troquem livros entre si, visualizem livros disponíveis e gerenciem suas próprias bibliotecas.

Em conclusão, o desenvolvimento do programa BookSwap foi um processo desafiador, mas gratificante. Através da aplicação de boas práticas de desenvolvimento de software e da realização de testes, conseguimos criar um software de qualidade que atende às necessidades básicas dos usuários. A experiência reforçou a importância dos testes unitários no desenvolvimento de software e destacou o valor de uma arquitetura de software bem projetada.

REFERÊNCIAS

JUnit 5. JUnit 5: The Programmer's Guide. Disponível em: https://junit.org/junit5/docs/current/user-guide/. Acesso em: 01/04/2024.

MOCKITO. **Mockito Framework**. Disponível em: https://site.mockito.org/. Acesso em: 20/04/2024.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. Design Patterns: **Elements of Reusable Object-Oriented Software**. Addison-Wesley Professional, 1994.

ORACLE. **Documentação oficial da Oracle para Java**. Disponível em: https://docs.oracle.com/en/java/. Acesso em: 06/04/2024.

JORGENSEN, P. C. **Software Testing: A Craftsman's Approach**. CRC Press, 4ª edição, 2
0
1