**Verificação e Validação de Software I – Trabalho 1**

**Antônio Marcos de Oliveira Pereira**

**PUCRS**

**Introdução**

Este relatório apresenta o desenvolvimento de um sistema de controle de estacionamento para centros comerciais, que visa calcular tarifas de forma precisa e eficiente. O sistema é orientado a objetos, com uma estrutura organizada de classes que encapsula entidades, serviços e aplicação. A funcionalidade central do sistema é a classe CalculaTarifa, responsável por calcular o valor a ser pago pelos usuários com base no tempo de permanência e nas condições específicas, como a categoria de cliente e o período de estacionamento.

**Testes Realizados**

**1. deveLancarExcecaoParaTicketNulo**

Este teste verifica se o método valorTicket lança uma IllegalArgumentException quando um ticket nulo é passado como parâmetro. O teste utiliza a técnica de teste de exceções, que assegura que o sistema responda adequadamente a entradas inválidas. Essa abordagem é fundamental para garantir que a aplicação não falhe silenciosamente em situações inesperadas. O objetivo é assegurar que a lógica de validação na classe CalculoValor esteja correta, prevenindo que entradas nulas provoquem falhas em outros pontos do sistema.

**2. deveLancarExcecaoParaDataEntradaNula**

Este teste verifica se uma IllegalArgumentException é lançada quando a data de entrada do ticket é nula. A validação de entradas nulas é uma técnica comum em programação defensiva, que ajuda a evitar comportamentos inesperados do sistema. O teste confirma que a aplicação está protegida contra dados incompletos. O objetivo é garantir que a lógica de negócios não processe tickets com informações faltantes, contribuindo para a integridade dos dados e a confiança no sistema.

**3. deveLancarExcecaoParaDataSaidaNula**

Este teste assegura que uma IllegalArgumentException seja lançada quando a data de saída do ticket é nula. Assim como no teste anterior, este teste utiliza a validação de entradas nulas para proteger o sistema contra dados inválidos. Essa técnica é essencial para a criação de um software robusto e confiável. O teste visa garantir que o sistema não permita que tickets com dados ausentes sejam processados, evitando cálculos incorretos e resultados inesperados.

**4. deveLancarExcecaoParaSaidaAntesDaEntrada**

Este teste verifica se o método lança uma IllegalArgumentException quando a data de saída é anterior à data de entrada.A validação de regras de negócio é uma prática importante que assegura que a lógica do sistema esteja alinhada com as expectativas do usuário. Esse teste é uma aplicação dessa técnica, garantindo que o sistema siga as regras definidas para o funcionamento do estacionamento.O objetivo é evitar que dados conflitantes sejam aceitos pelo sistema, assegurando que os usuários não enfrentem problemas com entradas e saídas incoerentes. Isso contribui para a experiência do usuário e a integridade do sistema.

**5. calcularTarifaCom15MinutosDeCortesia**

Este teste avalia se o sistema retorna um valor correto de 0,0 para um ticket cuja duração é de 15 minutos, que está dentro do período de cortesia. O teste utiliza a abordagem de teste de unidade, que valida o comportamento de pequenas partes do código em isolamento. Isso é essencial para garantir que cada parte do sistema funcione como esperado. O objetivo é assegurar que o sistema reconheça corretamente o período de cortesia, evitando cobranças indevidas e promovendo a satisfação do usuário.

**6. calcularTarifaNormalAteUmaHora**

Este teste verifica se a tarifa é calculada corretamente para um ticket que permanece por 30 minutos. Este teste também segue a técnica de teste de unidade, focando na validação de cálculos específicos dentro da lógica de tarifação.

O objetivo é garantir que o sistema aplique a tarifa padrão corretamente para o uso até uma hora, assegurando a precisão nas cobranças.

**7. calcularTarifaParaUmaHora**

Este teste confirma que a tarifa calculada para um ticket de uma hora é a mesma que a tarifa padrão. Assim como os testes anteriores, este teste faz uso da técnica de teste de unidade para avaliar a lógica de tarifação do sistema. O objetivo é garantir que o sistema mantenha consistência nas tarifas para períodos de uso definidos, promovendo a confiabilidade do sistema de cobrança.

**8. calcularTarifaParaPernoite**

Este teste verifica se a tarifa para um ticket que abrange uma pernoite é calculada corretamente. O teste utiliza a técnica de teste de unidade para verificar a lógica de cobrança para períodos mais longos, assegurando que o sistema lide corretamente com tarifas diferenciadas. O objetivo é garantir que a tarifa de pernoite seja aplicada corretamente, proporcionando um sistema de cobrança justo e transparente.

**9. calcularTarifaParaMultiplosDiasPernoite**

Este teste avalia se o sistema calcula corretamente a tarifa para um ticket que abrange várias pernoites. Assim como nos testes anteriores, este teste segue a técnica de teste de unidade, focando na validação de cálculos complexos dentro da lógica de tarifação. O objetivo é garantir que o sistema aplique corretamente as tarifas acumuladas para períodos prolongados, evitando erros de cobrança.

**10. aplicarDescontoVipParaTarifaNormal**

Este teste verifica se o desconto para clientes VIP é aplicado corretamente em uma tarifa normal. Este teste também se baseia na técnica de teste de unidade, validando a aplicação de regras de negócios específicas para diferentes tipos de clientes.O objetivo é assegurar que os benefícios para clientes VIP sejam implementados de forma correta, promovendo a lealdade do cliente.

**11. aplicarDescontoVipParaTarifaDePernoite**

Este teste confirma se o desconto para clientes VIP é aplicado corretamente em uma tarifa de pernoite. Assim como os testes anteriores, utiliza a técnica de teste de unidade, focando na validação de cálculos para diferentes tarifas. O objetivo é garantir que o sistema reconheça e aplique corretamente as tarifas diferenciadas para clientes VIP, assegurando que esses clientes recebam o tratamento adequado.

**12. deveCalcularTarifaRapidamente**

Este teste verifica se o cálculo da tarifa é realizado rapidamente, dentro de um limite de tempo específico. O teste de performance é fundamental para garantir que o sistema atenda às expectativas de eficiência e responsividade. Essa técnica ajuda a identificar potenciais gargalos de desempenho. O objetivo é assegurar que o sistema mantenha um tempo de resposta adequado, proporcionando uma experiência de usuário satisfatória.

**13. deveCalcularTarifaNosLimites**

Este teste avalia se a tarifa é calculada corretamente para tickets que estão exatamente nos limites de cortesia e de cobrança.**:** O teste de limites é uma técnica importante que assegura que o sistema funcione conforme esperado em situações que se aproximam dos limites estabelecidos. O objetivo é garantir que o sistema trate corretamente os casos em que as entradas estão nas extremidades de condições, evitando erros de cálculo.

**14. Teste de Cálculo de Tarifa com Técnica de Particionamento**

Este teste foi projetado para validar a lógica de cálculo de tarifas da classe CalculaTarifa utilizando a técnica de particionamento. O objetivo é garantir que o sistema funcione corretamente para diferentes categorias de entrada, dividindo-as em partições válidas e inválidas.

**E**sse teste assegura que o sistema lida corretamente com diferentes cenários de entrada, utilizando a técnica de particionamento para cobrir uma gama abrangente de condições. Ele garante não apenas a precisão nos cálculos, mas também a robustez da aplicação contra entradas inválidas, contribuindo para a qualidade geral do sistema de controle de estacionamento.

**Partições Válidas**

**Período Gratuito**: O teste verifica se o sistema retorna R$ 0,00 para uma estadia de 10 minutos, garantindo que o cálculo respeita o limite estabelecido para o período gratuito.

**Tarifa Normal**: Avalia se o valor correto de R$ 5,90 é retornado para uma estadia de 30 minutos, confirmando que o sistema aplica a tarifa normal adequadamente.

**Tarifa de Pernoite**: Testa se o valor de R$ 50,00 é corretamente calculado para um ticket que inicia à noite e termina pela manhã, assegurando que a aplicação considera as tarifas específicas para pernoites.

**Partições Inválidas**

**Ticket Nulo**: O teste garante que uma tentativa de calcular a tarifa de um ticket nulo resulta em uma exceção IllegalArgumentException, protegendo o sistema contra entradas inválidas.

**Data de Entrada Posterior à Data de Saída**: Verifica se o sistema lança uma exceção ao tentar calcular a tarifa com uma data de entrada que ocorre após a data de saída, reforçando a integridade dos dados e evitando cálculos incoerentes.

**15. deveCalcularTarifaComValoresAleatorios**

O objetivo deste teste é verificar se a classe CalculaTarifa é capaz de calcular corretamente o valor do ticket de estacionamento utilizando valores aleatórios de entrada e saída. Este teste é fundamental para assegurar que a lógica de cálculo se comporta conforme esperado em diferentes cenários de utilização.

Esse teste é crucial para validar a robustez e a precisão do sistema de cálculo de tarifas, garantindo que a implementação lide corretamente com uma variedade de cenários e que os resultados estejam de acordo com as regras de negócio estabelecidas

**Geração de Dados Aleatórios**: O teste utiliza a classe Random para gerar valores aleatórios que simulam entradas e saídas de veículos. Para cada iteração, uma data e hora de entrada é gerada no passado, enquanto a data e hora de saída é sempre posterior à de entrada. Isso garante que o ticket seja válido.

**Criação do Ticket**: Um objeto Ticket é criado com os horários de entrada e saída gerados aleatoriamente, e um estado booleano aleatório que determina se o cliente é VIP.

**Cálculo do Valor Esperado**: Para cada ticket, o valor esperado é calculado utilizando um método auxiliar (calcularTarifaEsperada), que considera as regras de tarifação definidas para o sistema de controle de estacionamento. O método leva em conta:

**Tempo de Estadia**: A tarifa varia com base na duração da estadia, oferecendo isenção para períodos curtos e aplicando tarifas diferenciadas para estadias mais longas.

**Desconto para Clientes VIP**: Um desconto de 30% é aplicado ao valor total se o cliente for VIP.

**Validação do Resultado**: O valor calculado pela instância da classe CalculaTarifa é comparado com o valor esperado utilizando o método assertEquals, com uma margem de erro de 0,01. Isso assegura que o valor retornado esteja dentro de um intervalo aceitável, considerando possíveis pequenas imprecisões nos cálculos.

**Problemas Resolvidos**

**Resolução de Bug: Validação de Dados de Entrada e Saída**

Durante o desenvolvimento do sistema de controle de estacionamento, identifiquei um bug crítico que ocorria quando os dados de entrada e saída do ticket de estacionamento não eram válidos. Especificamente, o problema surgia quando os campos de data e hora eram nulos ou quando a data e hora de saída eram anteriores à data e hora de entrada. Esses casos geravam comportamentos inesperados e resultados incorretos nos cálculos de tarifas, comprometendo a funcionalidade do sistema.

Texto

Descrição gerada automaticamente

**Diagnóstico do Problema**

Para abordar essa questão, decidi implementar um teste automatizado que verificasse a presença de dados nulos e a lógica temporal dos tickets. O teste consistia em criar tickets com as seguintes situações:

Texto

Descrição gerada automaticamente

**Data e Hora Nulas**: Criar tickets onde a data e hora de entrada ou saída eram nulos.**Saída Anterior à Entrada**: Criar tickets onde a data e hora de saída eram anteriores à data e hora de entrada.

Essas verificações foram essenciais para reproduzir o problema e entender as condições que levavam ao erro.

**Implementação da Solução**

Com os testes em vigor, fui capaz de identificar claramente os pontos onde o sistema falhava. Para resolver o problema, implementei uma série de validações dentro do método responsável pelo cálculo do valor do ticket. As principais alterações incluíram:

**Validação de Dados Nulos**: Adicionei uma verificação que lançava uma exceção específica (IllegalArgumentException) se a data e hora de entrada ou saída fossem nulas. Essa abordagem garante que o sistema não prossiga com dados inválidos, impedindo falhas posteriores no processamento.

**Verificação de Ordem Temporal**: Implementei outra validação que lançava uma exceção quando a data e hora de saída eram anteriores à data e hora de entrada. Essa medida assegura que todos os tickets sejam criados com informações coerentes e lógicas.

Tela preta com letras brancas

Descrição gerada automaticamente

**Resultados da Implementação**

Após a implementação das validações e a execução dos testes automatizados, o sistema apresentou uma melhoria significativa na robustez. Todos os casos de dados nulos e saídas anteriores à entrada foram corretamente identificados e tratados, evitando comportamentos indesejados. A adição das exceções não apenas facilitou o diagnóstico de problemas durante o uso do sistema, mas também melhorou a experiência do usuário ao fornecer mensagens de erro claras e informativas.

Esse processo de identificação e correção de bugs foi essencial para garantir a integridade do sistema de controle de estacionamento. Através de testes eficazes e validações apropriadas, consegui resolver um problema que impactava diretamente a funcionalidade e a confiabilidade do sistema. Essa experiência reforçou a importância de realizar testes abrangentes e implementar boas práticas de programação para evitar problemas semelhantes no futuro.

**Resolução de Bug: Validação de Ticket Nulo**

Durante o desenvolvimento do sistema de controle de estacionamento, identifiquei um erro crítico que ocorria quando um ticket nulo era passado para o método responsável pelo cálculo do valor a ser pago. Essa situação gerava exceções não tratadas e comportamentos inesperados, comprometendo a funcionalidade do sistema e a experiência do usuário.

Texto

Descrição gerada automaticamente

**Diagnóstico do Problema**

Para resolver essa questão, implementei um teste automatizado que verificasse se o ticket passado como argumento era nulo. O teste consistia em tentar calcular o valor do ticket usando um objeto Ticket que não havia sido instanciado. Essa abordagem permitiu identificar o ponto exato em que o sistema falhava e garantiu que a validação fosse feita de maneira eficaz :

Texto

Descrição gerada automaticamente

**Implementação da Solução**

Com o teste implementado, foquei na adição de uma validação no método responsável pelo cálculo da tarifa do ticket. As principais alterações realizadas foram:

**Validação de Ticket Nulo**: Adicionei uma verificação que lançava uma exceção específica (IllegalArgumentException) caso o ticket passado como parâmetro fosse nulo. Essa abordagem impede que o sistema prossiga com um objeto inválido, evitando erros subsequentes e garantindo que todas as operações relacionadas ao ticket sejam realizadas de forma segura.

**Mensagens de Erro Claras**: Implementei mensagens de erro descritivas nas exceções lançadas, facilitando a identificação do problema e proporcionando feedback claro para os desenvolvedores e usuários que interagem com o sistema.

Texto

Descrição gerada automaticamente

**Resultados da Implementação**

Após a implementação da validação para tickets nulos e a execução dos testes automatizados, a robustez do sistema foi significativamente aprimorada. O tratamento de tickets nulos foi corretamente identificado e tratado, evitando comportamentos inesperados. A inclusão das exceções não apenas melhorou a estabilidade do sistema, mas também resultou em uma experiência mais fluida para o usuário ao fornecer mensagens de erro compreensíveis.

A correção deste erro destacou a importância de realizar validações rigorosas e de implementar testes automatizados que cubram cenários críticos, como a verificação de objetos nulos. A experiência adquirida durante esse processo não apenas melhorou a integridade do sistema de controle de estacionamento, mas também ressaltou a relevância de manter boas práticas de programação para prevenir problemas semelhantes no futuro.

**Conclusão**

O sistema de controle de estacionamento desenvolvido atendeu aos requisitos propostos e se destacou pela eficiência e confiabilidade em suas operações. A validação foi realizada por meio de testes unitários abrangentes, utilizando JUnit, que garantiram a precisão no cálculo das tarifas e a robustez do sistema ao lidar com entradas inválidas. Essa etapa foi fundamental para identificar e corrigir falhas potenciais, assegurando que cada componente funcione conforme o esperado.

A implementação de testes com JUnit não apenas elevou a qualidade do software, mas também proporcionou uma base sólida para futuras manutenções e aprimoramentos. Essa prática é essencial para garantir a longevidade do sistema, permitindo que ele se adapte a novas demandas e desafios.

A escolha de uma abordagem orientada a objetos foi decisiva, pois facilitou a organização do código e a modularização das funcionalidades. Isso resultou em uma estrutura que favorece a manutenção e a escalabilidade, preparando o sistema para futuras implementações e melhorias.

Em resumo, o projeto não apenas cumpriu seu propósito inicial, mas também se mostrou capaz de evoluir e se adaptar, garantindo sua relevância no contexto do controle de estacionamento. O caminho está aberto para novas funcionalidades e aprimoramentos que certamente irão beneficiar os usuários e as operações do sistema.