Trabalho
Qualidade e Teste de Software

Alunos:
Antônio Ferreira
Bruno Mota de Mello
Tales Piotrowski
Darah Leite

Plano de Teste

1 INTRODUÇÃO

Este documento apresenta o Plano de Teste do projeto de Ponto de Venda (PDV) desenvolvido em Java. O foco é validar funcionalidades específicas relacionadas à gestão de vendas, usuários, caixas e parcelas, bem como garantir a correção lógica de componentes de serviço através de testes unitários. Serão detalhadas as estratégias, o escopo, os objetivos, as responsabilidades, a metodologia e os entregáveis relacionados às atividades de teste planejadas para esta fase inicial do projeto.

1.1 ESCOPO

O escopo deste Plano de Teste abrange as funcionalidades e componentes do sistema PDV que serão testados, com foco nas áreas atribuídas aos membros do grupo.

1.1.1 No escopo

Os seguintes módulos, componentes e funcionalidades do sistema PDV estão incluídos no escopo de teste:

- **Testes Unitários em Classes de Serviço:**

  Serão testadas unitariamente classes de serviço específicas que contêm lógica de negócio relevante, utilizando frameworks de teste para Java (JUnit). As classes e os responsáveis são:

- `PagarParcelaService` - Responsável: **Bruno**
- `VendaService` - Responsável: **Darah**

- `GrupoUsuarioService` - Responsável: **Vinicius**
- `CaixaService` - Responsável: **Thales**

O foco será em métodos dentro dessas classes que implementam regras de negócio, validações ou orquestram chamadas a múltiplos componentes, excluindo métodos que são meros wrappers para operações CRUD simples de repositório.

Testes Manuais em Funcionalidades Específicas:

Será testada manualmente pelo menos uma funcionalidade completa por membro do grupo, utilizando casos de teste projetados em documento de texto. As funcionalidades e os responsáveis são:

- Funcionalidades relacionadas à gestão de parcelas a pagar (ex: listagem de parcelas abertas) - Responsável: **Bruno**
- Fluxo completo de criação e fechamento de uma venda (incluindo adição/remoção de itens e simulação de pagamento) - Responsável: **Darah**
- Funcionalidades relacionadas à gestão de grupos de usuários e permissões (ex: adicionar permissão a um grupo) - Responsável: **Vinicius**
- Fluxo de abertura e fechamento de caixa - Responsável: **Tales**

1.1.2 Fora do escopo

As seguintes funcionalidades e aspectos do sistema PDV NÃO estão incluídos no escopo de teste :

- Integração com hardware real (impressoras fiscais, leitores de código de barras, máquinas de cartão).
- Módulos de retaguarda complexos não listados no escopo (relatórios gerenciais avançados, gestão de fornecedores, etc.).
- Testes de performance, carga ou stress.
- Testes de segurança.
- Testes de usabilidade extensivos.
- Testes unitários em classes que representam apenas entidades de dados (classes como `Venda`, `Caixa`, `GrupoUsuario`, `PagarParcela` em si, a menos que contenham lógica complexa além de getters/setters).
- Testes unitários em métodos de Repositório (`ParcelaRepository`, `VendaRepository`, `GrupoUsuarioRepository`, `CaixaRepository`) que são apenas operações CRUD ou consultas simples.
- Tratamento de erros de comunicação com serviços externos ou falhas de hardware.

- Funcionalidades de outros módulos não explicitamente listados no escopo (ex: gestão de produtos completa, gestão de clientes completa).

1.2 OBJETIVOS DE QUALIDADE

Os objetivos de qualidade a serem alcançados com as atividades de teste são:

- Verificar se as classes de serviço `PagarParcelaService`, `VendaService`, `GrupoUsuarioService` e `CaixaService` executam sua lógica de negócio corretamente para os métodos selecionados para teste unitário.
- Garantir que as funcionalidades de alto nível atribuídas para testes manuais operem conforme o esperado, simulando o uso real pelo usuário.
- Identificar e documentar o maior número possível de defeitos nas classes de serviço e funcionalidades testadas.
- Validar que os casos de teste (unitários e manuais) projetados cobrem os principais fluxos e cenários das partes do sistema no escopo.

1.3 PAPÉIS E RESPONSABILIDADES

As responsabilidades pelas atividades de teste serão distribuídas entre os membros da equipe da seguinte forma, focando nas classes e funcionalidades atribuídas:

- **Bruno:** Responsável principal pelo projeto e implementação dos testes unitários na classe `PagarParcelaService` e pelo projeto e execução dos testes manuais nas funcionalidades de gestão de parcelas a pagar.
- **Darah:** Responsável principal pelo projeto e implementação dos testes unitários na classe `VendaService` e pelo projeto e execução dos testes manuais no fluxo completo de venda.
- **Vinicius:** Responsável principal pelo projeto e implementação dos testes unitários na classe `GrupoUsuarioService` e pelo projeto e execução dos testes manuais nas funcionalidades de gestão de grupos de usuários e permissões.
- **Thales:** Responsável principal pelo projeto e implementação dos testes unitários na classe `CaixaService` e pelo projeto e execução dos testes manuais no fluxo de abertura e fechamento de caixa. Também responsável pela compilação e organização dos artefatos de teste (Plano de Teste, relatórios).

Todos os membros do grupo são responsáveis por reportar os defeitos encontrados e colaborar na análise e correção, se aplicável.

2. METODOLOGIA DE TESTE

2.1 VISÃO GERAL

A metodologia de teste adotada segue uma abordagem iterativa e incremental, alinhada com o desenvolvimento do projeto. Os testes unitários serão focados em validar a lógica de negócio das classes de serviço isoladamente, enquanto os testes manuais validarão o fluxo de usuário nas funcionalidades de alto nível.

2.2 FASES DE TESTE

As fases de teste incluem:

1. **Projeto dos Casos de Teste:** Definição dos cenários e passos para testes unitários (para as classes de serviço atribuídas) e manuais (para as funcionalidades atribuídas).
2. **Implementação dos Testes Unitários:** Codificação dos testes automatizados para as classes de serviço selecionadas.
3. **Execução dos Testes Unitários:** Rodar os testes automatizados e analisar os resultados.
4. **Execução dos Testes Manuais:** Seguir os passos definidos nos casos de teste manuais e registrar os resultados.
5. **Relatório de Defeitos:** Documentar quaisquer desvios encontrados entre o comportamento esperado e o real.
6. **Relatório de Execução:** Compilar os resultados gerais da execução dos testes.

2.3 TRIAGEM DE ERROS

Os defeitos encontrados durante a execução dos testes serão documentados em um relatório de erros compartilhados (utilizando um documento de texto ou planilha). A triagem inicial será feita pelos membros do grupo para classificar a severidade do defeito e determinar se ele deve ser corrigido nesta entrega ou postergado.

2.4 CRITÉRIOS DE SUSPENSÃO E REQUISITOS DE RETOMADA

- **Critérios de Suspensão:** Os testes manuais em uma funcionalidade específica podem ser suspensos se um defeito de alta severidade (que impeça a continuidade dos testes nessa funcionalidade) for encontrado. Testes unitários podem ser pausados se dependências críticas estiverem quebradas.

- **Requisitos de Retomada:** Os testes suspensos podem ser retomados assim que os defeitos bloqueadores forem corrigidos e uma nova versão do sistema/componente estiver disponível para teste, após uma verificação rápida da correção (teste de confirmação).

2.5 COMPLETUDE DO TESTE

Os critérios para considerar as atividades de teste como completas são:

- Todos os casos de teste unitários planejados para as classes de serviço no escopo foram implementados e executados.
- Todos os casos de teste manuais planejados para as funcionalidades no escopo foram executados.
- Todos os defeitos encontrados foram devidamente documentados no relatório de erros.
- Os relatórios de execução de testes unitários e manuais foram gerados.

2.6 ATIVIDADES DO PROJETO, ESTIMATIVAS E CRONOGRAMA

As principais atividades de teste em um cronograma estimado são:

Cronograma de Testes Unitários – 06/05 a 19/05

Data	Atividade
06/05	Revisar responsabilidades e dependências de todas as services
07/05	Planejar os cenários de teste para PagarParcelaService
08/05	Implementar testes unitários para PagarParcelaService (casos principais)
09/05	Implementar testes de exceções e bordas para PagarParcelaService
10/05	Planejar e estruturar testes para VendaService
11/05	Implementar testes unitários para VendaService (fluxos principais)

12/05	Testar exceções e condições alternativas em VendaService
13/05	Planejar testes para GrupoUsuarioService
14/05	Implementar testes unitários para GrupoUsuarioService
15/05	Planejar e iniciar testes para CaixaService
16/05	Implementar testes para CaixaService (fluxos principais e mocks)
17/05	Finalizar testes de borda/exceção para CaixaService
18/05	Rodar todos os testes, revisar mocks/stubs, ajustes finos
19/05	Revisar cobertura, nomeação, clareza, gerar documentação e commit final

3 ENTREGÁVEIS DE TESTE

Os artefatos de teste que serão gerados e entregues são:

- **Plano de Teste:** Este documento.
- **Código dos Testes Unitários:** Os arquivos `.java` contendo os testes unitários implementados para `PagarParcelaService`, `VendaService`, `GrupoUsuarioService` e `CaixaService` (parte do código-fonte original).
- **Relatório de Execução dos Testes Unitários:** Relatório gerado pela ferramenta de teste unitário (ex: saída do console do JUnit ou relatório HTML).
- **Casos de Teste Manuais:** Documento (texto ou planilha) descrevendo os passos, dados de entrada e resultados esperados para cada cenário de teste manual das funcionalidades atribuídas.

- **Relatório de Execução dos Testes Manuais:** Documento (texto ou planilha) registrando os resultados da execução de cada caso de teste manual (Passou/Falhou) e observações.
- **Relatório de Defeitos:** Documento (texto ou planilha) listando os defeitos encontrados, sua descrição, severidade e status.

4 NECESSIDADES DE RECURSOS E AMBIENTE

4.1 FERRAMENTAS DE TESTE

As ferramentas a serem utilizadas para as atividades de teste incluem:

- **Ferramenta de Teste Unitário:** JUnit (ou outra framework de teste unitário para Java) será utilizada para testar as classes de serviço.
- **Framework de Mocking (Opcional, mas recomendado para Unitários):** Mockito ou similar, para simular o comportamento das dependências (como os Repositórios e outros Services) ao testar as classes de Serviço isoladamente.
- **Ambiente de Desenvolvimento Integrado (IDE):** Eclipse, IntelliJ IDEA ou similar, utilizado para escrever e executar o código do sistema e dos testes unitários.
- **Ferramentas de Documentação:** Documentos de texto (ex: Google Docs, Microsoft Word) ou planilhas eletrônicas (ex: Google Sheets, Microsoft Excel) serão utilizados para documentar os casos de teste manuais, relatórios de execução e relatórios de defeitos. **Conforme especificado, a ferramenta TestLink NÃO será utilizada nesta entrega.**

4.2 AMBIENTE DE TESTE

O ambiente mínimo necessário para executar os testes é:

- **Sistema Operacional:** Windows 10 ou superior, macOS, ou distribuição Linux compatível com Java.
- **Java Development Kit (JDK):** Versão 8 ou superior.
- **Memória RAM:** Mínimo de 4GB (recomendado 8GB ou mais).
- **Espaço em Disco:** Espaço suficiente para o código-fonte do projeto, a IDE e as dependências.
- **Software Adicional:** A IDE escolhida (Eclipse, IntelliJ, etc.).

Relatórios de teste

1. Bruno - `PagarParcelaService` (Testes Unitários) e Gestão de Parcelas a Pagar (Testes Manuais)

- **Testes Unitários (**):
- **Método a testar:** `cadastrar(Double vltotal, Double vlrestante, int quitado, Timestamp cadastro, LocalDate vencimento, Pagar pagar)`
- **Cenário:** Cadastro de uma nova parcela com valores válidos.
- **Entrada:** Valores positivos para `vltotal`, `vlrestante`, `quitado=0`, datas válidas, objeto `Pagar` válido.
- **Resultado Esperado:** O método `parcelas.geraParcela` (do `PagarParcelaRespository` mockado) é chamado exatamente uma vez com os parâmetros corretos. Nenhuma exceção é lançada.

Cenário: Cadastro com `vltotal` zero ou negativo.

- **Entrada:** `vltotal = 0.0` ou `-10.0`, outros parâmetros válidos.
- **Resultado Esperado:** Caso o `vltotal seja z`ero ou negativo, o sistema abrirá uma excessão para esta compra.

Cenário: Cadastro onde o repositório lança uma exceção.

- **Entrada:** Parâmetros válidos, mas configure o mock de `parcelas.geraParcela` para lançar uma `Exception`.
- **Resultado Esperado:** O método `cadastrar` captura a exceção, imprime o stack trace e lança uma nova `RuntimeException`.

Método a testar: `lista(PagarParcelaFilter filter, Pageable pageable)`

- **Cenário:** Listar parcelas sem filtro de nome.
- **Entrada:** `filter` com nome nulo ou vazio, `pageable` válido.
- **Resultado Esperado:** O método `parcelas.listaOrdenada(pageable)` é chamado.

Cenário: Listar parcelas com filtro de nome.

- **Entrada:** `filter` com nome preenchido (ex: "Fornecedor X"), `pageable` válido.
- **Resultado Esperado:** O método `parcelas.listaOrdenada(filter.getNome(), pageable)` é chamado com o nome correto.

Cenário: Listar parcelas quando o repositório retorna uma página vazia.

- **Entrada:** Qualquer filtro/pageable. Configure o mock do repositório para retornar `Page.empty()`.
- **Resultado Esperado:** O método `lista` retorna uma página vazia.

Testes Manuais (Gestão de Parcelas a Pagar):

- **Funcionalidade:** Visualizar lista de parcelas a pagar abertas.
- **Pré-condição:** Existem parcelas cadastradas no sistema, algumas abertas e algumas quitadas.
- **Casos de Teste:**
- **CTM_PAR_001: Acessar a tela de listagem de parcelas a pagar.**
- **Passos:** Navegar até a funcionalidade de "Parcelas a Pagar".
- **Resultado Esperado:** A tela de listagem é exibida, mostrando por padrão apenas as parcelas com status "Aberto". As informações exibidas (valor total, restante, vencimento, etc.) estão corretas.

CTM_PAR_002: Filtrar parcelas por nome de fornecedor/pagador.

- **Passos:** Na tela de listagem, digitar o nome de um fornecedor/pagador existente no campo de filtro e aplicar o filtro.
- **Resultado Esperado:** A lista é atualizada para mostrar apenas as parcelas associadas a esse fornecedor/pagador.

CTM_PAR_003: Verificar o total de despesas abertas exibido.

- **Passos:** Acessar a tela de listagem de parcelas a pagar.
- **Resultado Esperado:** O valor total de despesas abertas exibido na tela corresponde à soma dos valores restantes de todas as parcelas com status "Aberto".

2. Darah - `VendaService` (Testes Unitários) e Fluxo Completo de Venda (Testes Manuais)

- **Testes Unitários ()**:
- **Método a testar:** `fechaVenda(Long venda, Long pagamentotipo, Double vlprodutos, Double desconto, Double acrescimo, String[] vlParcelas, String[] titulos)`
- **Cenário:** Tentar fechar venda que já está fechada.
- **Entrada:** `venda` (código de uma venda que está com `situacao = VendaSituacao.FECHADA`), outros parâmetros podem ser quaisquer valores válidos ou nulos, pois a validação da situação ocorre no início do método. Configure o mock de `vendas.findByCodigoEquals(venda)` para retornar um objeto `Venda` onde `isAberta()` retorna `false`.
- **Resultado Esperado:** O método lança uma `RuntimeException` com a mensagem "venda fechada". Nenhum dos outros serviços (receberServ, parcelas, lancamentos, vendas.fechaVenda, produtos.movimentaEstoque) deve ser chamado.

3. Vinicius - `GrupoUsuarioService` (Testes Unitários) e Gestão de Grupos de Usuários/Permissões (Testes Manuais)

- **Testes Unitários ()**:
- **Método a testar:** `merge(GrupoUsuario grupoUsuario, RedirectAttributes attributes)`
- **Cenário:** Cadastrar um novo grupo de usuário com sucesso.
- **Entrada:** `grupoUsuario` com `codigo = null`, outros campos preenchidos (nome, descrição). Configure o mock de `grupousuarios.save` para não fazer nada ou retornar o objeto salvo. Configure o mock de `attributes.addFlashAttribute` para não fazer nada.
- **Resultado Esperado:** O método `grupousuarios.save` é chamado exatamente uma vez com o objeto `grupoUsuario`. O método `attributes.addFlashAttribute` é chamado com a chave "mensagem" e a mensagem de sucesso. Nenhuma exceção é lançada.

Cenário: Atualizar um grupo de usuário existente com sucesso.

- **Entrada:** `grupoUsuario` com `codigo` preenchido (código de um grupo existente), outros campos preenchidos (nome, descrição). Configure o mock de `grupousuarios.update` para não fazer nada. Configure o mock de `attributes.addFlashAttribute` para não fazer nada.
- **Resultado Esperado:** O método `grupousuarios.update` é chamado exatamente uma vez com os parâmetros corretos (nome, descrição, código). O método `attributes.addFlashAttribute` é chamado com a chave "mensagem" e a mensagem de atualização. Nenhuma exceção é lançada.

Cenário: Tentar cadastrar um grupo e ocorrer um erro no repositório.

- **Entrada:** `grupoUsuario` com `codigo = null`. Configure o mock de `grupousuarios.save` para lançar uma `Exception`. Configure o mock de `System.out.println` para não imprimir no console de teste.
- **Resultado Esperado:** O método `grupousuarios.save` é chamado. A exceção é capturada e impressa no console (simulado pelo mock). Nenhuma exceção é relançada pelo método `merge`.

Método a testar: `remove(Long codigo, RedirectAttributes attributes)`

- **Cenário:** Remover um grupo que NÃO possui usuários vinculados.
- **Entrada:** `codigo` de um grupo. Configure o mock de `grupousuarios.grupoTemUsuaio(codigo)` para retornar `0`. Configure o mock de `grupousuarios.deleteById(codigo)` para não fazer nada.
- **Resultado Esperado:** O método `grupousuarios.grupoTemUsuaio` é chamado. O método `grupousuarios.deleteById` é chamado exatamente uma vez com o código correto. Retorna a string "redirect:/grupousuario".

Cenário: Tentar remover um grupo que POSSUI usuários vinculados.

- **Entrada:** `codigo` de um grupo. Configure o mock de `grupousuarios.grupoTemUsuaio(codigo)` para retornar `1` (ou qualquer valor > 0). Configure o mock de `attributes.addFlashAttribute` para não fazer nada.
- **Resultado Esperado:** O método `grupousuarios.grupoTemUsuaio` é chamado. O método `grupousuarios.deleteById` NÃO é chamado. O método `attributes.addFlashAttribute` é chamado com a chave "mensagemErro" e a mensagem de erro. Retorna a string "redirect:/grupousuario/" + codigo.

Método a testar: `addPermissao(Long codgrupo, Long codpermissao)`

- **Cenário:** Adicionar uma permissão a um grupo onde ela AINDA NÃO existe.
- **Entrada:** `codgrupo`, `codpermissao`. Configure o mock de `grupousuarios.grupoTemPermissao(codgrupo, codpermissao)` para retornar `0`. Configure o mock de `grupousuarios.addPermissao(codgrupo, codpermissao)` para não fazer nada.
- **Resultado Esperado:** O método `grupousuarios.grupoTemPermissao` é chamado. O método `grupousuarios.addPermissao` é chamado exatamente uma vez com os códigos corretos. Retorna a string "Permissao adicionada com sucesso".

Cenário: Tentar adicionar uma permissão a um grupo onde ela JÁ existe.

- **Entrada:** `codgrupo`, `codpermissao`. Configure o mock de `grupousuarios.grupoTemPermissao(codgrupo, codpermissao)` para retornar `1` (ou qualquer valor > 0).
- **Resultado Esperado:** O método `grupousuarios.grupoTemPermissao` é chamado. O método `grupousuarios.addPermissao` NÃO é chamado. O método lança uma `RuntimeException` com a mensagem "Esta permissão já esta adicionada a este grupo".

Testes Manuais (Gestão de Grupos de Usuários/Permissões):

- **Funcionalidade:** Adicionar uma permissão a um grupo de usuário.
- **Pré-condição:** Existem grupos de usuários e permissões cadastrados no sistema. O usuário logado tem permissão para gerenciar grupos e permissões.
- **Casos de Teste:**
- **CTM_GRU_001: Adicionar uma permissão válida a um grupo.**
- **Passos:**
- Acessar a tela de gestão de grupos de usuários.
- Selecionar um grupo existente.
- Na seção de permissões do grupo, selecionar uma permissão que ainda não está associada a ele.
- Clicar no botão para adicionar a permissão.
- **Resultado Esperado:** A permissão é adicionada à lista de permissões do grupo. Uma mensagem de sucesso é exibida.

CTM_GRU_002: Tentar adicionar uma permissão que já está associada ao grupo.

- **Passos:**
- Acessar a tela de gestão de grupos de usuários.
- Selecionar um grupo existente.
- Na seção de permissões do grupo, selecionar uma permissão que *já* está associada a ele.
- Clicar no botão para adicionar a permissão.
- **Resultado Esperado:** O sistema exibe uma mensagem de erro informando que a permissão já existe para o grupo. A permissão não é duplicada na lista.

CTM_GRU_003: Remover uma permissão de um grupo.

- **Passos:**
- Acessar a tela de gestão de grupos de usuários.
- Selecionar um grupo existente que possua permissões associadas.
- Na lista de permissões do grupo, selecionar uma permissão.
- Clicar no botão para remover a permissão.
- **Resultado Esperado:** A permissão é removida da lista de permissões do grupo. Uma mensagem de sucesso é exibida.

4. Thales - `CaixaService` (Testes Unitários) e Fluxo de Abertura/Fechamento de Caixa (Testes Manuais)

- **Testes Unitários (**):
- **Método a testar:** `cadastro(Caixa caixa)`
- **Cenário:** Cadastrar um novo caixa do tipo CAIXA com valor de abertura positivo, sem caixa anterior aberto.
- **Entrada:** Objeto `Caixa` com `codigo = null`, `tipo = CaixaTipo.CAIXA`, `valor_abertura = 100.00`, `descricao` vazia ou preenchida. Configure mocks para `caixas.caixaIsAberto()` retornar `false`, `Aplicacao.getInstancia().getUsuarioAtual()` retornar um usuário, `usuarios.buscaUsuario` retornar um `Usuario`, `caixas.save` não fazer nada, `lancamentos.lancamento` não fazer nada.

- **Resultado Esperado:** O método `caixas.caixaIsAberto` é chamado. O método `caixas.save` é chamado com o objeto `Caixa` preenchido (descrição padrão "Caixa diário" se vazia, usuário, data_cadastro). O método `lancamentos.lancamento` é chamado para o lançamento de saldo inicial. Retorna o código do caixa salvo.

Cenário: Cadastrar um novo caixa do tipo CAIXA com valor de abertura zero, sem caixa anterior aberto.

- **Entrada:** Objeto `Caixa` com `codigo = null`, `tipo = CaixaTipo.CAIXA`, `valor_abertura = 0.0` ou `null`. Configure mocks similar ao cenário anterior.
- **Resultado Esperado:** O método `caixas.caixaIsAberto` é chamado. O método `caixas.save` é chamado. O método `lancamentos.lancamento` NÃO é chamado. O `valor_total` do objeto `Caixa` antes de salvar é definido para 0.0. Retorna o código do caixa salvo.

Cenário: Tentar cadastrar um caixa do tipo CAIXA quando já existe um caixa anterior aberto.

- **Entrada:** Objeto `Caixa` com `tipo = CaixaTipo.CAIXA`. Configure o mock de `caixas.caixaIsAberto()` para retornar `true`.
- **Resultado Esperado:** O método `caixas.caixaIsAberto` é chamado. O método lança uma `RuntimeException` com a mensagem "Existe caixa de dias anteriores em aberto, favor verifique". Nenhum outro método de repositório ou serviço é chamado.

Cenário: Cadastrar um novo caixa do tipo BANCO com dados de agência/conta.

- **Entrada:** Objeto `Caixa` com `codigo = null`, `tipo = CaixaTipo.BANCO`, `valor_abertura` (pode ser 0 ou positivo), `descricao` (vazia ou preenchida), `agencia` e `conta` com caracteres não numéricos (ex: "123-X", "456.Y"). Configure mocks similar ao cenário de cadastro de CAIXA, mas `caixas.caixaIsAberto()` pode retornar true (não bloqueia BANCO).
- **Resultado Esperado:** O método `caixas.caixaIsAberto` é chamado (mas não bloqueia). O método `caixas.save` é chamado com o objeto `Caixa` onde `agencia` e `conta` foram limpos de caracteres não numéricos. Se `valor_abertura > 0`, `lancamentos.lancamento` é chamado. Retorna o código do caixa salvo.

Método a testar: `fechaCaixa(Long caixa, String senha)`

- **Cenário:** Fechar um caixa aberto com senha correta.

- **Entrada:** `caixa` (código de um caixa aberto), `senha` (senha correta do usuário logado). Configure mocks para `Aplicacao.getInstancia().getUsuarioAtual()` retornar um usuário, `usuarios.buscaUsuario` retornar um `Usuario`, `BCryptPasswordEncoder.matches(senha, usuario.getSenha())` retornar `true`, `caixas.findById(caixa)` retornar um `Optional` contendo um `Caixa` aberto (data_fechamento nula), `caixas.save` não fazer nada.
- **Resultado Esperado:** A senha é validada. O método `caixas.findById` é chamado. O método `caixas.save` é chamado com o objeto `Caixa` atualizado (data_fechamento e valor_fechamento preenchidos). Retorna a string "Caixa fechado com sucesso".

Cenário: Tentar fechar um caixa aberto com senha incorreta.

- **Entrada:** `caixa` (código de um caixa aberto), `senha` (senha incorreta). Configure mocks para `BCryptPasswordEncoder.matches(senha, usuario.getSenha())` retornar `false`.
- **Resultado Esperado:** A senha é validada como incorreta. O método `caixas.findById` NÃO é chamado. Retorna a string "Senha incorreta, favor verifique".

Cenário: Tentar fechar um caixa que já está fechado.

- **Entrada:** `caixa` (código de um caixa já fechado), `senha` (senha correta). Configure mocks para `BCryptPasswordEncoder.matches` retornar `true`, `caixas.findById(caixa)` retornar um `Optional` contendo um `Caixa` já fechado (data_fechamento preenchida).
- **Resultado Esperado:** A senha é validada. O método `caixas.findById` é chamado. O método lança uma `RuntimeException` com a mensagem "Caixa já esta fechado". O método `caixas.save` NÃO é chamado.

Cenário: Tentar fechar um caixa sem informar a senha.

- **Entrada:** `caixa` (qualquer código), `senha` vazia ("").
- **Resultado Esperado:** O método verifica que a senha está vazia. Retorna a string "Favor, informe a senha". Nenhuma validação de senha ou busca no repositório é feita.

Testes Manuais (Fluxo de Abertura/Fechamento de Caixa):

- **Funcionalidade:** Abrir um novo caixa diário.
- **Pré-condição:** Não existe nenhum caixa diário aberto para dias anteriores. O usuário logado tem permissão para abrir caixa.
- **Casos de Teste:**

- **CTM_CAI_001: Abrir caixa diário com valor de abertura positivo.**
- **Passos:**
- Acessar a funcionalidade de "Abrir Caixa".
- Selecionar o tipo "Caixa Diário".
- Informar um valor positivo para "Valor de Abertura" (ex: 150.00).
- Informar uma descrição (opcional).
- Confirmar a abertura.
- **Resultado Esperado:** O sistema exibe uma mensagem de sucesso. Um novo caixa é registrado como aberto com o valor e descrição informados. Um lançamento de "Saldo Inicial" é registrado para este caixa com o valor de abertura.

CTM_CAI_002: Abrir caixa diário com valor de abertura zero.

- **Passos:**
- Acessar a funcionalidade de "Abrir Caixa".
- Selecionar o tipo "Caixa Diário".
- Informar 0.00 ou deixar o campo "Valor de Abertura" vazio.
- Informar uma descrição (opcional).
- Confirmar a abertura.
- **Resultado Esperado:** O sistema exibe uma mensagem de sucesso. Um novo caixa é registrado como aberto com valor de abertura 0.00. Nenhum lançamento de "Saldo Inicial" é registrado.

CTM_CAI_003: Tentar abrir caixa diário quando já existe um caixa diário aberto de dia anterior.

- **Pré-condição Adicional:** Existe um caixa do tipo "Caixa Diário" registrado com data de abertura em um dia anterior e sem data de fechamento.
- **Passos:**
- Acessar a funcionalidade de "Abrir Caixa".
- Selecionar o tipo "Caixa Diário".
- Informar valor de abertura e descrição.
- Confirmar a abertura.
- **Resultado Esperado:** O sistema exibe uma mensagem de erro informando que existe um caixa anterior aberto e que ele deve ser fechado primeiro. Um novo caixa NÃO é aberto.

Funcionalidade: Fechar um caixa diário.

Pré-condição: Existe um caixa diário aberto. O usuário logado tem permissão para fechar caixa e conhece a senha correta.

Casos de Teste:

1. **CTM_CAI_004: Fechar caixa diário com senha correta.**
2. **Passos:**
3. Acessar a funcionalidade de "Fechar Caixa".
4. Selecionar o caixa diário aberto.
5. Informar a senha correta do usuário logado.
6. Confirmar o fechamento.

Resultado Esperado: O sistema exibe uma mensagem de sucesso. O caixa selecionado é marcado como fechado com a data/hora e o valor total final registrados.

CTM_CAI_005: Tentar fechar caixa diário com senha incorreta.

- **Passos:**
- Acessar a funcionalidade de "Fechar Caixa".
- Selecionar o caixa diário aberto.
- Informar uma senha incorreta.
- Confirmar o fechamento.
- **Resultado Esperado:** O sistema exibe uma mensagem de erro informando que a senha está incorreta. O caixa permanece aberto.

CTM_CAI_006: Tentar fechar caixa diário que já está fechado.

- **Pré-condição Adicional:** Existe um caixa do tipo "Caixa Diário" que já foi fechado anteriormente.
- **Passos:**
- Acessar a funcionalidade de "Fechar Caixa".
- Selecionar um caixa que já está fechado.
- Informar a senha.
- Confirmar o fechamento.
- **Resultado Esperado:** O sistema exibe uma mensagem de erro informando que o caixa já está fechado. O status do caixa não muda.