

TDD: Transforme Seu Código com Testes de Alto Desempenho



Wellington Lara

Índice

1. Introdução ao Test-Driven Development (TDD)
2. Configurando o Ambiente: Java e Spring Boot
3. O Ciclo do TDD: Red, Green, Refactor
4. Escrevendo Seu Primeiro Teste: "Hello, World!" com TDD
5. Testes de Unidade Simples com Spring Boot
6. Introduzindo Mocking e Test Doubles
7. Trabalhando com Testes de Integração
8. Refatoração: Melhorando Seu Código com Segurança
9. Lidando com Exceções e Testes Negativos
10. TDD em Projetos Reais: Casos de Uso
11. Vantagens do TDD no Desenvolvimento de Software

Capítulo 1: Introdução ao Test-Driven Development (TDD)

O que é TDD?

Test-Driven Development (TDD) é uma abordagem de desenvolvimento de software que enfatiza a criação de testes automatizados antes da implementação do código de produção. Ao adotar TDD, os desenvolvedores escrevem testes que descrevem o comportamento desejado de uma funcionalidade antes de escrever o próprio código que implementa essa funcionalidade. Isso resulta em um ciclo contínuo de desenvolvimento que garante a criação de um código mais robusto e menos propenso a bugs.

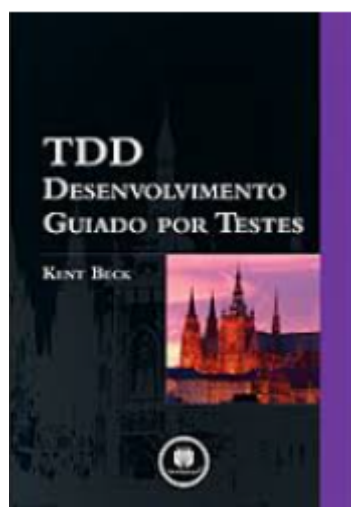
Por que Usar TDD?

A principal motivação para usar TDD é a melhoria da qualidade do código. Com testes escritos antecipadamente, os desenvolvedores são forçados a pensar sobre os requisitos e o

design do software de forma mais profunda. Além disso, TDD facilita a refatoração, já que os testes garantem que mudanças no código não quebrem funcionalidades existentes. Outros benefícios incluem uma documentação viva na forma de testes e um ciclo de feedback mais rápido durante o desenvolvimento.

História e Evolução

O conceito de TDD foi popularizado por Kent Beck com o lançamento do seu livro "Test-Driven Development by Example" em 2003. Desde então, TDD tem se tornado uma prática comum em metodologias ágeis e é amplamente adotado por equipes de desenvolvimento que buscam aumentar a qualidade e a manutenibilidade de seu código.



Como Funciona o TDD?

O TDD segue um ciclo bem definido conhecido como "Red, Green, Refactor":

- **Red:** Escreva um teste que falhe, pois a funcionalidade ainda não foi implementada.
 - **Green:** Escreva o código mínimo necessário para fazer o teste passar.
 - **Refactor:** Melhore o código garantindo que os testes continuem a passar.
-

Capítulo 2: Configurando o Ambiente: Java e Spring Boot

Instalando o Java

Para começar com TDD em Java, o primeiro passo é instalar o JDK (Java Development Kit). Você pode baixar a versão mais recente do JDK no site oficial da Oracle ou optar por distribuições alternativas como OpenJDK.

Configurando o Spring Boot

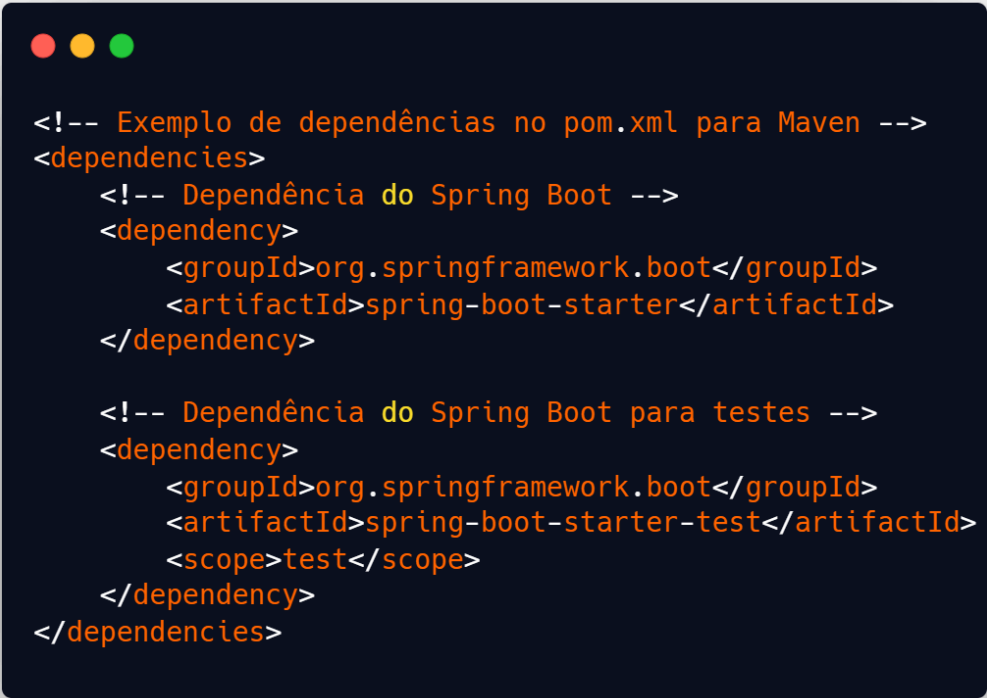
Spring Boot é uma extensão do Spring Framework que simplifica o processo de criação de aplicativos Java. Para configurar um novo projeto Spring Boot, você pode usar o Spring Initializr, uma ferramenta online que gera um projeto básico com todas as dependências necessárias. Basta acessar <https://start.spring.io> e selecionar as opções desejadas para seu projeto.

Ferramentas Necessárias

Além do JDK e do Spring Boot, você precisará de um IDE (Integrated Development Environment) para escrever e executar seu código. IntelliJ IDEA, Eclipse e VS Code são opções populares. Também é recomendável instalar o Maven ou Gradle, que são ferramentas de automação de build usadas pelo Spring Boot.

Configuração Inicial

Depois de gerar seu projeto Spring Boot, importe-o para o seu IDE. O Spring Boot já vem configurado com uma estrutura básica de pastas e um arquivo `pom.xml` (para Maven) ou `build.gradle` (para Gradle) que gerencia as dependências do projeto. A seguir, vamos adicionar dependências específicas para testes como JUnit e Mockito.



```
<!-- Exemplo de dependências no pom.xml para Maven -->
<dependencies>
  <!-- Dependência do Spring Boot -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>

  <!-- Dependência do Spring Boot para testes -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```


Capítulo 3: O Ciclo do TDD:

Red, Green, Refactor

Entendendo o Ciclo

O ciclo "Red, Green, Refactor" é o coração do TDD. Esse processo cíclico garante que o desenvolvimento avance de forma incremental e controlada, sempre guiado pelos testes.

Fase Red

Na fase "Red", você escreve um teste que ainda não pode passar porque a funcionalidade que ele verifica ainda não foi implementada. Isso ajuda a clarificar os requisitos e define o objetivo imediato.



```
// Exemplo de teste na fase Red
@Test
public void shouldReturnHelloWorld() {
    String result = helloService.sayHello();
    assertEquals("Hello, World!", result);
}
```

Fase Green

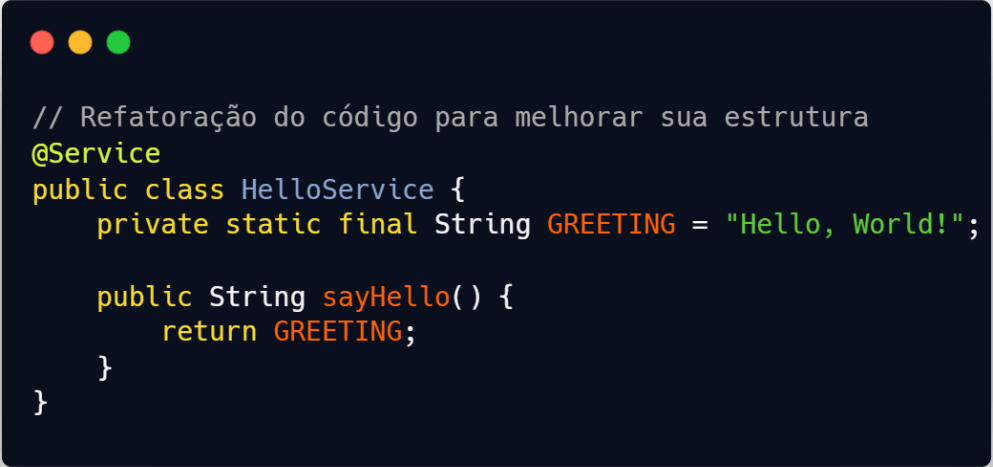
Na fase "Green", você escreve apenas o suficiente de código para fazer o teste passar. O foco aqui é na implementação mínima que satisfaz o teste.



```
// Implementação mínima para passar o teste
@Service
public class HelloService {
    public String sayHello() {
        return "Hello, World!";
    }
}
```

Fase Refactor

A fase "Refactor" envolve melhorar o código escrito, garantindo que ele seja limpo e eficiente, sem alterar seu comportamento. Os testes existentes ajudam a garantir que as mudanças não quebrem a funcionalidade.



```
// Refatoração do código para melhorar sua estrutura
@Service
public class HelloService {
    private static final String GREETING = "Hello, World!";

    public String sayHello() {
        return GREETING;
    }
}
```

Iteração Contínua

O ciclo se repete para cada nova funcionalidade ou ajuste necessário. Essa abordagem incremental permite que os desenvolvedores construam software de alta qualidade com confiança.

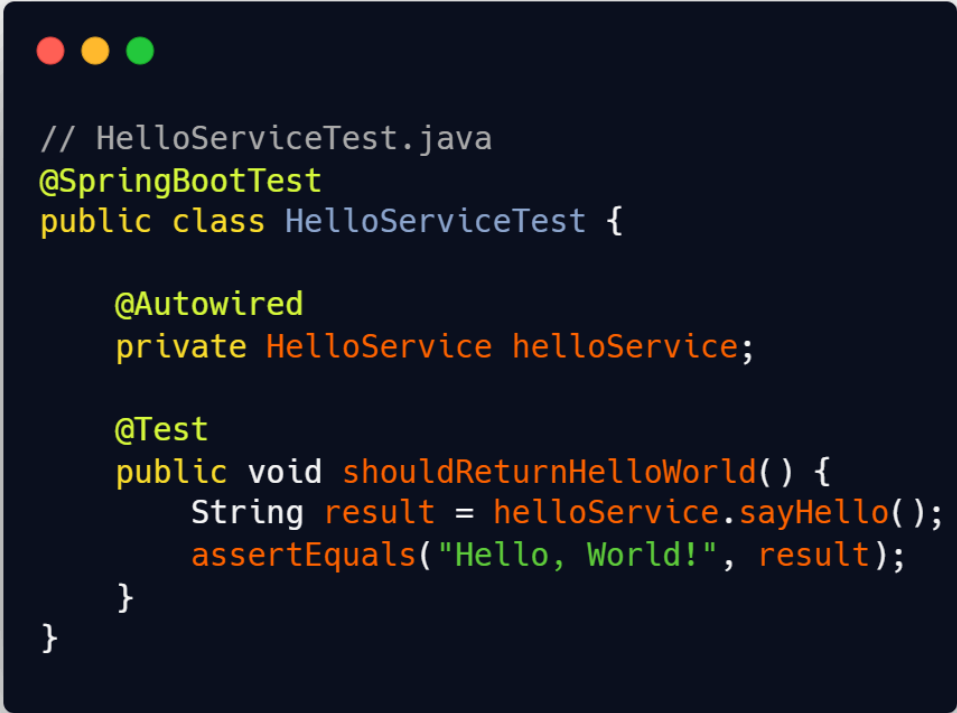
Capítulo 4: Escrevendo Seu Primeiro Teste: "Hello, World!" com TDD

Preparação do Projeto

Vamos criar um simples serviço Spring Boot que retorna a string "Hello, World!". Primeiro, crie uma nova classe de serviço `HelloService` no seu projeto.

Teste Inicial

Comece escrevendo um teste que define o comportamento desejado.



```
// HelloServiceTest.java
@SpringBootTest
public class HelloServiceTest {

    @Autowired
    private HelloService helloService;

    @Test
    public void shouldReturnHelloWorld() {
        String result = helloService.sayHello();
        assertEquals("Hello, World!", result);
    }
}
```

Implementação para Passar o Teste


Com o teste escrito, implemente a funcionalidade mínima para fazer o teste passar.



```
// HelloService.java
@Service
public class HelloService {
    public String sayHello() {
        return "Hello, World!";
    }
}
```

Refatoração

Depois de garantir que o teste passa, verifique se há oportunidades para melhorar o código sem alterar seu comportamento.



```
// Refatoração do HelloService
@Service
public class HelloService {
    private static final String MESSAGE = "Hello, World!";

    public String sayHello() {
        return MESSAGE;
    }
}
```

Conclusão

Parabéns! Você escreveu seu primeiro teste usando TDD. Ao seguir este processo, você garantiu que a funcionalidade fosse implementada corretamente desde o início e que está preparada para futuras mudanças e melhorias.

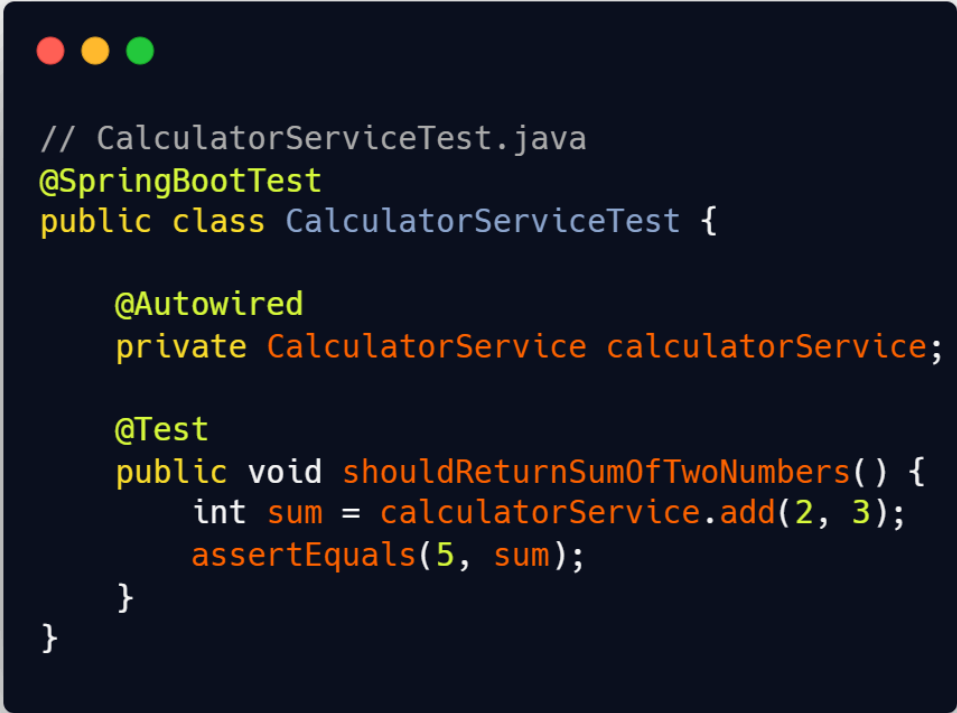
Capítulo 5: Testes de Unidade Simples com Spring Boot

Definindo Testes de Unidade

Testes de unidade verificam partes isoladas do código, geralmente métodos individuais, para garantir que funcionem conforme o esperado. Em um ambiente Spring Boot, esses testes geralmente envolvem classes de serviço ou componentes individuais.

Exemplo de Teste de Unidade

Vamos criar um serviço que calcula a soma de dois números. Primeiro, escreva o teste.



```
// CalculatorServiceTest.java
@SpringBootTest
public class CalculatorServiceTest {

    @Autowired
    private CalculatorService calculatorService;

    @Test
    public void shouldReturnSumOfTwoNumbers() {
        int sum = calculatorService.add(2, 3);
        assertEquals(5, sum);
    }
}
```

Implementação do Serviço

Agora, implemente a funcionalidade mínima para passar o teste.



```
// CalculatorService.java
@Service
public class CalculatorService {
    public int add(int a, int b) {
        return a + b;
    }
}
```

Refatoração e Expansão

Refatore o código conforme necessário e adicione novos testes para outras operações matemáticas, como subtração e multiplicação.



```
// CalculatorService.java (Refatorado)
@Service
public class CalculatorService {
    public int add(int a, int b) {
        return a + b;
    }

    public int subtract(int a, int b) {
        return a - b;
    }

    public int multiply(int a, int b) {
        return a * b;
    }
}
```

Testes Expandidos



```
// CalculatorServiceTest.java (Expansão)
@SpringBootTest
public class CalculatorServiceTest {

    @Autowired
    private CalculatorService calculatorService;

    @Test
    public void shouldReturnSumOfTwoNumbers() {
        int sum = calculatorService.add(2, 3);
        assertEquals(5, sum);
    }

    @Test
    public void shouldReturnDifferenceOfTwoNumbers() {
        int result = calculatorService.subtract(5, 3);
        assertEquals(2, result);
    }

    @Test
    public void shouldReturnProductOfTwoNumbers() {
        int product = calculatorService.multiply(2, 3);
        assertEquals(6, product);
    }
}
```

Conclusão

Os testes de unidade são fundamentais para garantir que cada parte do seu código funcione corretamente de forma isolada.

Usando TDD, você desenvolve funcionalidades confiáveis e prontas para futuras melhorias.

Capítulo 6: Introduzindo Mocking e Test Doubles

O Que São Mocks?

Mocks são objetos que imitam o comportamento de objetos reais de forma controlada. Eles são usados em testes para substituir dependências reais e isolar o código sob teste. Isso é especialmente útil quando o código a ser testado depende de recursos externos como bancos de dados ou serviços externos.

Mockito

Mockito é uma das bibliotecas mais populares para criar mocks em testes Java. Ele permite criar, configurar e verificar objetos mock de maneira simples e intuitiva.

Exemplo de Uso do Mockito

Vamos supor que temos um serviço `UserService` que depende de um repositório `UserRepository`. Queremos testar `UserService` sem precisar de uma implementação real do repositório.

```
// UserServiceTest.java
@SpringBootTest
public class UserServiceTest {

    @Mock
    private UserRepository userRepository;


    @InjectMocks
    private UserService userService;

    @BeforeEach
    public void setUp() {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void shouldReturnUserWhenUserExists() {
        User mockUser = new User(1L, "John Doe");
        Mockito.when(userRepository.findById(1L)).thenReturn(Optional.of(mockUser));

        User user = userService.getUserById(1L);
        assertEquals("John Doe", user.getName());
    }
}
```

Implementação do Serviço



```
// UserService.java
@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public User getUserById(Long id) {
        return userRepository.findById(id).orElseThrow(() -> new UserNotFoundException("User not found"));
    }
}
```

Vantagens do Mocking

Usar mocks permite que você:

- Isole o código sob teste de suas dependências externas.
- Teste diferentes cenários, incluindo casos extremos e de erro.
- Reduza a complexidade e o tempo de execução dos testes.

Conclusão

Mocks são ferramentas poderosas no arsenal de um desenvolvedor TDD. Eles permitem que você escreva testes mais eficazes e abrangentes, garantindo que o código funcione corretamente em uma variedade de situações.

Capítulo 7: Trabalhando com Testes de Integração

O Que São Testes de Integração?

Testes de integração verificam a interação entre diferentes partes do sistema, garantindo que elas funcionem corretamente juntas. Em um contexto Spring Boot, isso pode incluir a integração com bancos de dados, serviços REST e outros componentes do sistema.

Configurando Testes de Integração

Para configurar testes de integração em um projeto Spring Boot, você pode usar a anotação `@SpringBootTest`, que carrega todo o contexto da aplicação.

```
// ApplicationTest.java
@SpringBootTest
public class ApplicationTest {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void shouldReturnDefaultMessage() {
        ResponseEntity<String> response = restTemplate.getForEntity("/", String.class);
        assertEquals(HttpStatus.OK, response.getStatusCode());
        assertEquals("Hello, World!", response.getBody());
    }
}
```

Exemplo com Banco de Dados

Para testar a integração com um banco de dados, você pode usar um banco de dados em memória como H2.

```
// UserRepositoryTest.java
@SpringBootTest
public class UserRepositoryTest {

    @Autowired
    private UserRepository userRepository;

    @Test
    public void shouldFindUserById() {
        User user = new User(null, "Jane Doe");
        user = userRepository.save(user);

        Optional<User> foundUser = userRepository.findById(user.getId());
        assertTrue(foundUser.isPresent());
        assertEquals("Jane Doe", foundUser.get().getName());
    }
}
```

Melhorando a Manutenção com Testes de Integração

Os testes de integração são mais abrangentes que os testes de unidade, mas também são mais lentos e complexos. É importante equilibrar a quantidade de testes de integração e unidade para garantir uma cobertura de testes eficiente.

Conclusão

Testes de integração são essenciais para garantir que diferentes componentes do sistema funcionem corretamente em conjunto. Combinados com TDD, eles ajudam a construir sistemas robustos e confiáveis.

Capítulo 8: Refatoração:

Melhorando Seu Código com

Segurança

Importância da Refatoração

Refatoração é o processo de melhorar a estrutura interna do código sem alterar seu comportamento externo. Com TDD, a refatoração se torna mais segura, pois os testes garantem que as mudanças não quebrem a funcionalidade existente.

Técnicas Comuns de Refatoração

Algumas técnicas comuns incluem:

- Renomear variáveis e métodos para nomes mais claros.
- Extrair métodos para reduzir a complexidade.
- Remover duplicação de código.
- Simplificar expressões e lógicas condicionais.

Exemplo de Refatoração

Vamos refatorar um método que calcula descontos aplicados a um pedido.

```
// Antes da refatoração
public BigDecimal applyDiscount(Order order) {
    BigDecimal discount = BigDecimal.ZERO;
    if (order.getTotal().compareTo(BigDecimal.valueOf(100)) > 0) {
        discount = order.getTotal().multiply(BigDecimal.valueOf(0.1));
    }
    return discount;
}
```

Após a Refatoração

```
// Depois da refatoração
public BigDecimal applyDiscount(Order order) {
    return order.getTotal().compareTo(BigDecimal.valueOf(100)) > 0
        ? order.getTotal().multiply(BigDecimal.valueOf(0.1))
        : BigDecimal.ZERO;
}
```

Garantindo a Qualidade com Testes

Antes de refatorar, certifique-se de que há testes cobrindo o comportamento atual. Depois de refatorar, execute os testes para garantir que o comportamento não mudou.

Conclusão

Refatoração contínua é crucial para manter o código limpo e de alta qualidade. Com TDD, você pode refatorar com confiança, sabendo que os testes cobrem o comportamento esperado do seu código.

Capítulo 9: Lidando com Exceções e Testes Negativos

A Importância dos Testes Negativos

Testes negativos verificam como o sistema se comporta em condições adversas ou inesperadas. Eles são essenciais para garantir que o sistema seja robusto e lide corretamente com erros.

Escrevendo Testes para Exceções

Vamos criar um serviço que lança uma exceção quando um usuário não é encontrado.

```

// UserServiceTest.java
@SpringBootTest
public class UserServiceTest {

    @Autowired
    private UserService userService;

    @Test
    public void shouldThrowExceptionWhenUserNotFound() {
        assertThrows(UserNotFoundException.class, () -> {
            userService.getUserById(99L);
        });
    }
}

```

Implementação do Serviço

```

// UserService.java
@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public User getUserById(Long id) {
        return userRepository.findById(id)
            .orElseThrow(() -> new UserNotFoundException("User not found"));
    }
}

```


Testes de Validação

Além de exceções, você deve testar validações de entrada para garantir que o sistema lida corretamente com dados inválidos.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains a Java test method for UserServiceTest.java. The code uses JUnit annotations like @Test and assertEquals, and Mockito's assertThrows to verify that an exception is thrown when creating a user with invalid data (null name and empty email).

```
// UserServiceTest.java
@Test
public void shouldValidateUserInput() {
    User invalidUser = new User(null, "");
    Exception exception = assertThrows(ValidationException.class, () -> {
        userService.createUser(invalidUser);
    });
    assertEquals("Invalid user data", exception.getMessage());
}
```

Implementação da Validação



```
// UserService.java
public User createUser(User user) {
    if (user.getName() == null || user.getName().isEmpty()) {
        throw new ValidationException("Invalid user data");
    }
    return userRepository.save(user);
}
```

Conclusão

Testes negativos são fundamentais para construir sistemas resilientes. Ao antecipar e testar condições de erro, você garante que seu software lida corretamente com situações inesperadas, aumentando sua robustez.

Capítulo 10: TDD em Projetos Reais: Casos de Uso

Integração de TDD no Ciclo de Desenvolvimento

Integrar TDD em um projeto real requer disciplina e adaptação da equipe. É essencial seguir o ciclo "Red, Green, Refactor" rigorosamente e garantir que todos os desenvolvedores estejam comprometidos com a abordagem.

Caso de Uso: Aplicação de e-Commerce

Vamos considerar um projeto de e-commerce. TDD pode ser aplicado desde o início para garantir que todas as funcionalidades essenciais, como cadastro de produtos, processamento de pedidos e gestão de inventário, sejam implementadas de maneira robusta.

Exemplo: Processamento de Pedidos

Teste Inicial

```
// OrderServiceTest.java
@SpringBootTest
public class OrderServiceTest {

    @Autowired
    private OrderService orderService;

    @Test
    public void shouldProcessOrder() {
        Order order = new Order();
        order.addItem(new Item("Product1", 2, BigDecimal.valueOf(50)));

        Order processedOrder = orderService.processOrder(order);
        assertEquals(OrderStatus.PROCESSED, processedOrder.getStatus());
        assertEquals(BigDecimal.valueOf(100), processedOrder.getTotal());
    }
}
```

Implementação

```
// OrderService.java
@Service
public class OrderService {

    public Order processOrder(Order order) {
        order.setStatus(OrderStatus.PROCESSED);
        order.setTotal(order.getItems().stream()
            .map(item -> item.getPrice().multiply(BigDecimal.valueOf(item.getQuantity())))
            .reduce(BigDecimal.ZERO, BigDecimal::add));
        return order;
    }
}
```

Adaptação Contínua

Conforme o projeto cresce, é importante continuar aplicando TDD para novas funcionalidades e refatorar testes e código conforme necessário. Manter a disciplina é crucial para colher os benefícios a longo prazo.

Conclusão

Aplicar TDD em projetos reais requer prática e comprometimento. No entanto, os benefícios em termos de qualidade de código e facilidade de manutenção fazem com que o esforço valha a pena.

Capítulo 11: Vantagens do TDD no Desenvolvimento de Software

Melhoria na Qualidade do Código

TDD força os desenvolvedores a pensarem primeiro nos requisitos e comportamentos esperados antes de escrever o código de produção. Isso resulta em um design de software mais cuidadoso e robusto.

Facilidade na Refatoração

Com uma suíte de testes abrangente, a refatoração se torna segura e eficiente. Os testes garantem que as alterações no código não introduzam novos bugs.

Redução de Bugs

Ao escrever testes antes do código de produção, muitos bugs são evitados desde o início. Além disso, a execução contínua dos testes ajuda a detectar e corrigir erros rapidamente.

Feedback Rápido

Os testes automatizados fornecem feedback imediato sobre o impacto das mudanças no código. Isso acelera o ciclo de desenvolvimento e aumenta a confiança dos desenvolvedores nas suas alterações.

Documentação Viva

Os testes servem como uma documentação viva do comportamento do sistema. Eles descrevem claramente como o código deve se comportar em diferentes cenários, facilitando o entendimento e a manutenção do software.

Aumento da Produtividade

Embora escrever testes possa parecer um esforço adicional, a redução de bugs e a facilidade de manutenção resultam em um ganho de produtividade a longo prazo. Desenvolvedores

passam menos tempo corrigindo problemas e mais tempo implementando novas funcionalidades.

Conclusão

Adotar TDD traz inúmeros benefícios para o desenvolvimento de software. Além de melhorar a qualidade e a robustez do código, TDD facilita a manutenção e a evolução do sistema, proporcionando um retorno significativo sobre o investimento inicial em tempo e esforço. Ao incorporar TDD na sua prática diária, você estará construindo um software melhor e mais confiável.

Parabéns por completar este eBook sobre TDD! Esperamos que você tenha aprendido como aplicar TDD em seus projetos e visto os benefícios que essa abordagem pode trazer para o desenvolvimento de software. Continue praticando e aperfeiçoando suas habilidades para se tornar um desenvolvedor ainda mais eficiente e produtivo. Boa sorte!