Reuso de *Test Fixtures*

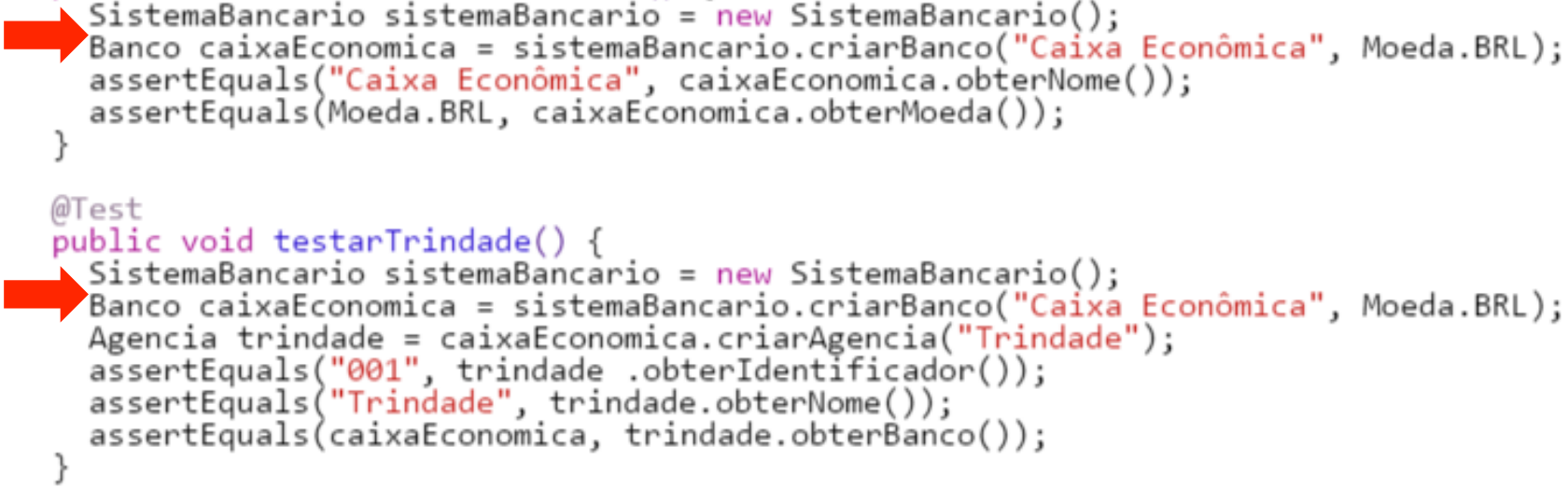
Lucas Pereira da Silva

Manutenibilidade dos testes

- Assim como o código de produção, o código de teste também precisa ser mantido, compreendido e ajustado.
- Reduzir a duplicação de código de teste auxilia na manutenibilidade dos testes.
- Mudanças na implementação afetam os testes.

Manutenibilidade dos testes

- Exemplo de código de teste com duplicação.



```
@Test
public void testarCaixaEconomica() {
  SistemaBancario sistemaBancario = new SistemaBancario();
  Banco caixaEconomica = sistemaBancario.criarBanco("Caixa Econômica", Moeda.BRL);
  assertEquals("Caixa Econômica", caixaEconomica.obterNome());
  assertEquals(Moeda.BRL, caixaEconomica.obterMoeda());
}

@Test
public void testarTrindade() {
  SistemaBancario sistemaBancario = new SistemaBancario();
  Banco caixaEconomica = sistemaBancario.criarBanco("Caixa Econômica", Moeda.BRL);
  Agencia trindade = caixaEconomica.criarAgencia("Trindade");
  assertEquals("001", trindade .obterIdentificador());
  assertEquals("Trindade", trindade.obterNome());
  assertEquals(caixaEconomica, trindade.obterBanco());
}
```

Etapas da execução de teste

1. Inicialização dos *test fixtures* (*fixture setup*).
2. Exercício do SUT.
3. Verificação das saídas do SUT.
4. Remoção dos *test fixtures* (*fixture teardown*).

```
@Test
public void testarCaixaEconomica() {
  SistemaBancario sistemaBancario = new SistemaBancario();
  Banco caixaEconomica = sistemaBancario.criarBanco("Caixa Econômica", Moeda.BRL);
  assertEquals("Caixa Econômica", caixaEconomica.obterNome());
  assertEquals(Moeda.BRL, caixaEconomica.obterMoeda());
}
```

Estratégias de *fixture setup*

- *Fresh fixture setup*.
- *Shared fixture construction*.

Inline setup

```
public class TesteBanco {

  @Test
  public void testarCaixaEconomica() {
    SistemaBancario sistemaBancario = new SistemaBancario();
    Banco caixaEconomica = sistemaBancario.criarBanco("Caixa Econômica", Moeda.BRL);
    assertEquals("Caixa Econômica", caixaEconomica.obterNome());
    assertEquals(Moeda.BRL, caixaEconomica.obterMoeda());
  }

}

public class TesteAgencia {

  @Test
  public void testarTrindade() {
    SistemaBancario sistemaBancario = new SistemaBancario();
    Banco caixaEconomica = sistemaBancario.criarBanco("Caixa Econômica", Moeda.BRL);
    Agencia trindade = caixaEconomica.criarAgencia("Trindade");
    assertEquals("001", trindade .obterIdentificador());
    assertEquals("Trindade", trindade .obterNome());
    assertEquals(caixaEconomica, trindade .obterBanco());
  }

}
```

Inline setup

- Boa compreensão da relação de causa e efeito entre *test fixtures* e saídas do SUT.
- Liberdade de organização das classes de teste.
- Causa duplicação de código de teste.

Implicit setup

```
public class TesteBancoAgencia {

  private Banco caixaEconomica;

  @Before
  public void configurar() {
    SistemaBancario sistemaBancario = new SistemaBancario();
    caixaEconomica = sistemaBancario.criarBanco("Caixa Econômica", Moeda.BRL);
  }

  @Test
  public void testarCaixaEconomica() {
    assertEquals("Caixa Econômica", caixaEconomica.obterNome());
    assertEquals(Moeda.BRL, caixaEconomica.obterMoeda());
  }

  @Test
  public void testarTrindade() {
    Agencia trindade = caixaEconomica.criarAgencia("Trindade");
    assertEquals("001", trindade .obterIdentificador());
    assertEquals("Trindade", trindade .obterNome());
    assertEquals(caixaEconomica, trindade .obterBanco());
  }

}
```

Implicit setup

- Promove o reuso de código de teste entre testes de uma mesma classe.
- Moderada compreensão da relação de causa e efeito entre *test fixtures* e saídas do SUT.
- Limita a organização das classes de teste.

Delegate setup

```
public class TesteBanco {

  @Test
  public void testarCaixaEconomica() {
    Banco caixaEconomica = Auxiliar.criarCaixaEconomica();
    assertEquals("Caixa Econômica", caixaEconomica.obterNome());
    assertEquals(Moeda.BRL, caixaEconomica.obterMoeda());
  }

}

public class TesteAgencia {

  @Test
  public void testarTrindade() {
    Banco caixaEconomica = Auxiliar.criarCaixaEconomica();
    Agencia trindade = Auxiliar.criarTrindade(caixaEconomica);
    assertEquals("001", trindade .obterIdentificador());
    assertEquals("Trindade", trindade .obterNome());
    assertEquals(caixaEconomica, trindade .obterBanco());
  }

}

public class Auxiliar {

  public static Banco criarCaixaEconomica() {
    SistemaBancario sistemaBancario = new SistemaBancario();
    return sistemaBancario.criarBanco("Caixa Econômica", Moeda.BRL);
  }

  public static Agencia criarTrindade(Banco banco) {
    return banco.criarAgencia("Trindade");
  }

}
```

Delegate setup

- Promove parcialmente o reuso de código de teste.
- Esconde detalhes não necessários para o teste.
- Liberdade de organização das classes de teste.
- Custo adicional para gerenciar classes e métodos auxiliares.
- Permite acesso à apenas um *test fixture*.
- Dificulta a compreensão da relação de causa e efeito entre *test fixtures* e saídas do SUT.

Problema

- Como permitir que classes de teste utilizem *test fixtures* definidos em uma ou mais classes de teste já existentes sem que, para isso, seja necessário alterar a estrutura das classes envolvidas?

Objetivo geral

- Definição de modelos que permitam a implementação de uma estratégia de *fixture setup* que promova o aumento do reuso de test fixtures e contribua para o desenvolvimento de testes com melhor manutenibilidade.

Proposta

- Observação: os *test fixtures* definidos em uma classe de teste podem levar o SUT exatamente para o estado necessário por testes de uma outra classe.
- Como reutilizar os *test fixtures* definidos na classe `TesteBanco` para a classe `TesteAgencia`?
- Modelo de dependência entre classes de teste.
- Estabelecer uma relação de dependência entre classes de teste.
- Classe **provedora**.
- Classe **consumidora**.

Reuso de código de *test fixture*

```
public class TesteBanco {

  private Banco caixaEconomica;

  @Before
  public void configurar() {
    SistemaBancario sistemaBancario = new SistemaBancario();
    caixaEconomica = sistemaBancario.criarBanco("Caixa Econômica", Moeda.BRL);
  }

  @Test
  public void testarCaixaEconomica() {
    assertEquals("Caixa Econômica", caixaEconomica.obterNome());
    assertEquals(Moeda.BRL, caixaEconomica.obterMoeda());
  }

}
```

Reuso de código de *test fixture*

```
@FixtureSetup(TesteBanco.class)
public class TesteAgencia {

  @Fixture private Banco caixaEconomica;

  private Agencia trindade;

  @Before
  public void configurar() {
    trindade = caixaEconomica.criarAgencia("Trindade");
  }

  @Test
  public void testarTrindade() {
    assertEquals("001", trindade .obterIdentificador());
    assertEquals("Trindade", trindade .obterNome());
    assertEquals(caixaEconomica, trindade .obterBanco());
  }

}
```

Reuso de código de *test fixture*

- Classe `TesteConta`.
- Os *test fixtures* definidos na classe `TesteAgencia` podem ser reutilizados pela classe `TesteConta` também?
- Transitividade das relações de dependência.

Reuso de código de *test fixture*

```
@FixtureSetup(TesteAgencia.class)
public class TesteConta {

  @Fixture private Agencia trindade;

  private Conta maria;

  @Before
  public void configurar() throws Exception {
    maria = trindade.criarConta("Maria");
  }

  @Test
  public void testarMaria() {
    assertEquals("0001-5", maria.obterIdentificador());
    assertEquals("Maria", maria.obterTitular());
    assertTrue(maria.calcularSaldo().zero());
    assertEquals(trindade, maria.obterAgencia());
  }

}
```

Reuso de código de *test fixture*

- `TesteDeposito` e `TesteDinheiro`.
- A classe `TesteDeposito` pode depender dos *test fixtures* definidos tanto na classe `TesteConta` quanto na classe `TesteDinheiro`?
- Múltiplas dependências.

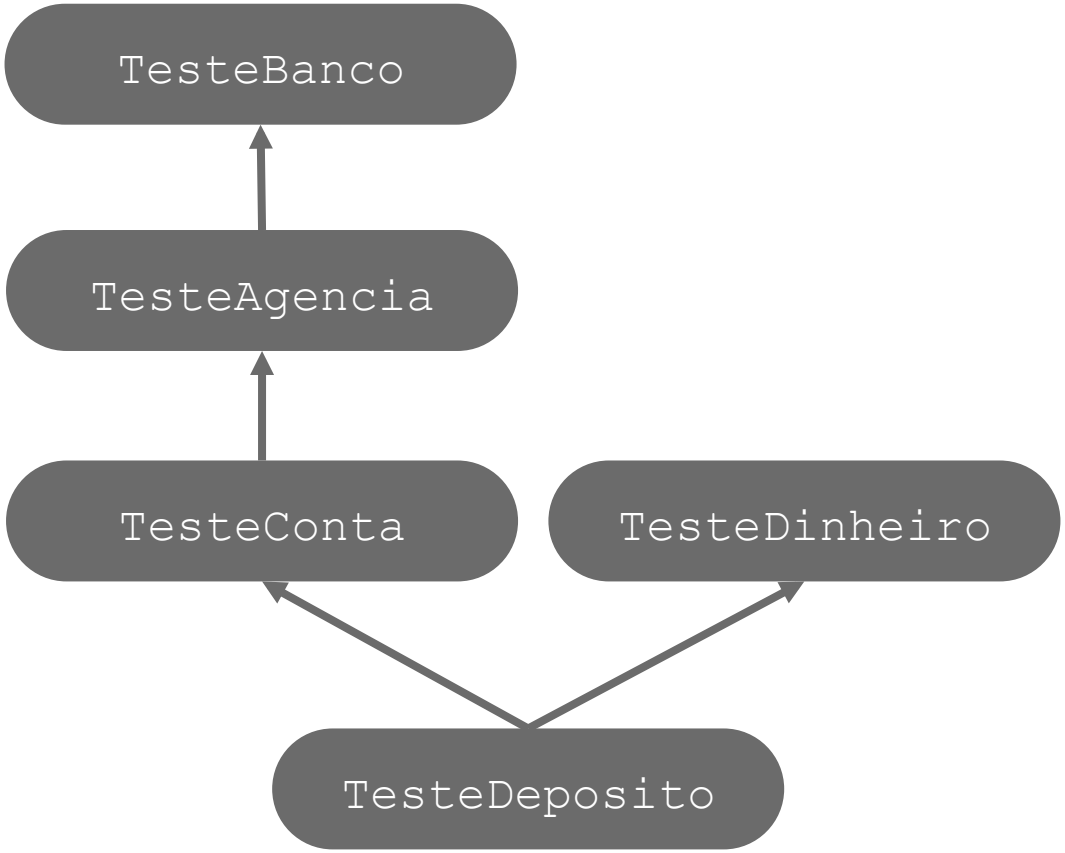
Reuso de código de *test fixture*

```
public class TesteDinheiro {

  private Dinheiro dezReais;

  @Before
  public void configurar() {
    dezReais = new Dinheiro(Moeda.BRL, 10, 0);
  }

  @Test
  public void testarDezReais() {
    assertEquals("10,00 BRL", dezReais.formatar());
    assertEquals(1000, dezReais.obterQuantiaEmEscala());
    assertEquals(Moeda.BRL, dezReais.obterMoeda());
  }

}
```

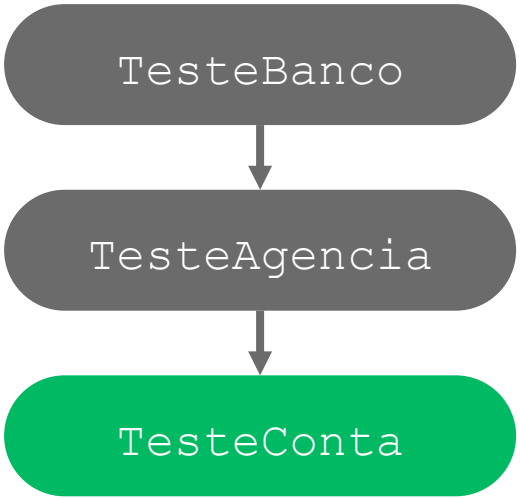
Reuso de código de *test fixture*

```
@FixtureSetup({
  TesteConta.class,
  TesteDinheiro.class
})
public class TesteDeposito {

  @Fixture private Conta maria;
  @Fixture private Dinheiro dezReais;

  @Test
  public void testarDeposito() {
    Operacao operacao = sistemaBancario.depositar(maria, dezReais);
    assertEquals(EstadosDeOperacao.SUCESSO, operacao.obterEstado());
    assertEquals(dezReais.positivo(), maria.calcularSaldo());
  }

}
```

Grafos de dependência e execução



Grafo de dependência



Grafo de execução

Reuso de código de *test fixture*

```
@FixtureSetup(TesteBanco.class)
public class TesteAgencia {

  @Fixture private Banco caixaEconomica;

  private Agencia trindade;

  @Before
  public void configurar() {
    trindade = caixaEconomica.criarAgencia("Trindade");
  }

  @Test
  public void testarTrindade() {
    assertEquals("001", trindade .obterIdentificador());
    assertEquals("Trindade", trindade .obterNome());
    assertEquals(caixaEconomica, trindade .obterBanco());
    trindade = null;
  }

}
```

Reuso de código de *test fixture*

```
@FixtureSetup(TesteAgencia.class)
public class TesteConta {

  @Fixture private Agencia trindade;

  private Conta maria;

  @Before
  public void configurar() throws Exception {
    maria = trindade.criarConta("Maria");
  }

  @Test
  public void testarMaria() {
    assertEquals("0001-5", maria.obterIdentificador());
    assertEquals("Maria", maria.obterTitular());
    assertTrue(maria.calcularSaldo().zero());
    assertEquals(trindade, maria.obterAgencia());
  }

}
```

Reuso de código de *test fixture*

```
@Unsafe
@Test
public void testarTrindade() {
  assertEquals("001", trindade .obterIdentificador());
  assertEquals("Trindade", trindade .obterNome());
  assertEquals(caixaEconomica, trindade .obterBanco());
  trindade = null;
}

@Safe
@Test
public void testarTrindade() {
  assertEquals("001", trindade .obterIdentificador());
  assertEquals("Trindade", trindade .obterNome());
  assertEquals(caixaEconomica, trindade .obterBanco());
}
```

Framework

- Desenvolvimento do *framework* Estória.
- Extensão do *framework* de testes JUnit.
- Adição das anotações apresentadas no exemplos para suportar a proposta deste trabalho: `@FixtureSetup`, `@Fixture`, `@Singular` e `@Safe`.
- Implementação de 38 classes.
- Desenvolvimento dirigido a testes (TDD).
- Implementação de 223 testes distribuídos através de 28 classes de teste.

Implementação do `Runner`

- Implementação do fluxo de execução.
- Classe `Estoria` sobrescreve o método `run` da classe `Runner`.