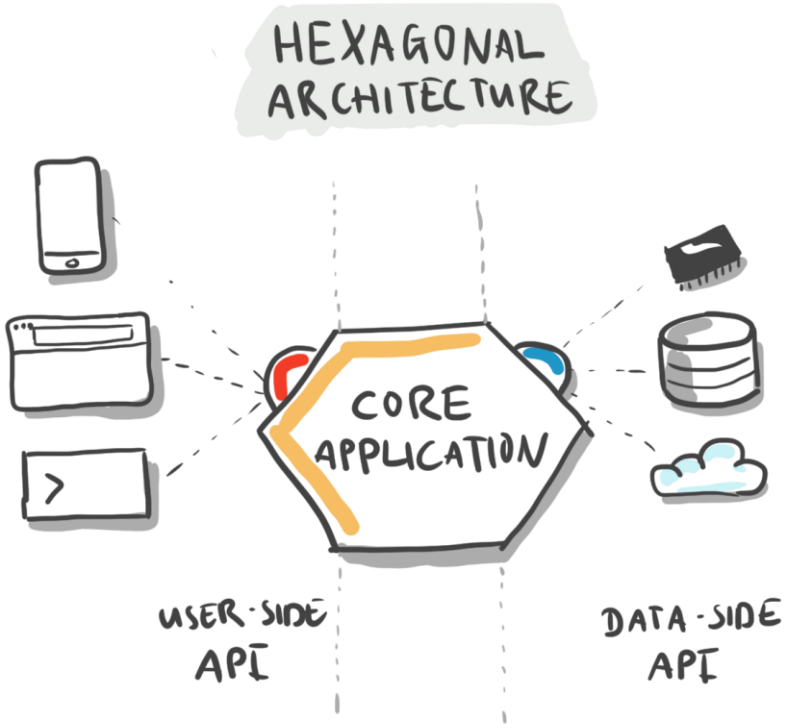
Apostila de Exercícios



Arquitetura Hexagonal Com Java

Fernando Franzini
fernandofranzini@gmail.com
https://fernandofranzini.wordpress.com/

Pré-requisitos

1. Ter uma JDK 12+ devidamente instalada e funcionando.
2. Ter IntelliJ Community 2018.1+ devidamente instalado e funcionando.
3. Faço sugestão de fazer meu curso de intelliJ – https://www.udemy.com/intellij-ide-para-desenvolvedores-java/

PROJETO HEXAGONAL

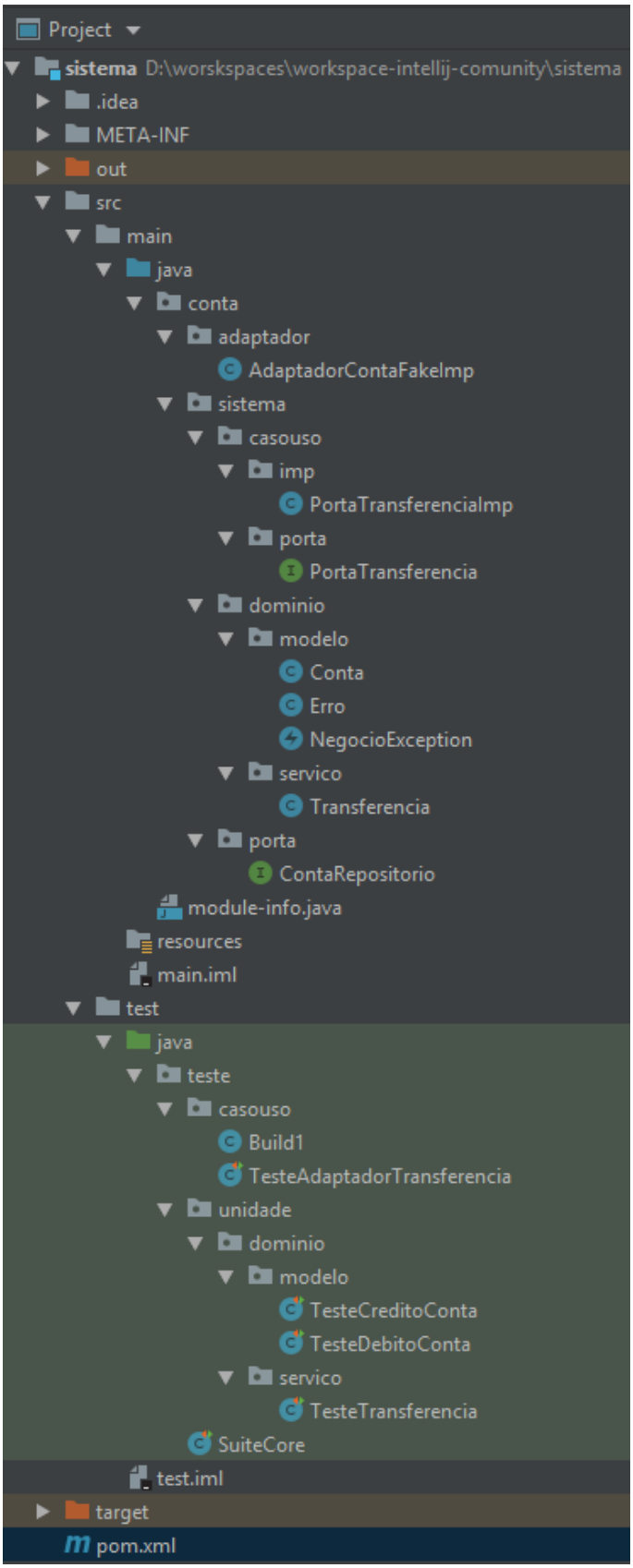
Criar um projeto do **centro** "hexagonal":

- Crie projeto chamado Maven "sistema".
- Configure as dependências no pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>sistema</groupId>
    <artifactId>sistema</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>12</maven.compiler.source>
        <maven.compiler.target>12</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- JUnit -->
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>5.4.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.platform</groupId>
            <artifactId>junit-platform-runner</artifactId>
            <version>1.4.1</version>
            <scope>test</scope>
        </dependency>

        <!-- CDI -->
        <dependency>
            <groupId>javax.enterprise</groupId>
            <artifactId>cdi-api</artifactId>
            <version>2.0</version>
        </dependency>

        <!-- Spring IoC -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.6.RELEASE</version>
        </dependency>

        <!-- Spring Transaction -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-tx</artifactId>
            <version>5.1.6.RELEASE</version>
        </dependency>

        <!-- Spring Test -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>5.1.6.RELEASE</version>
```

```
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

Criar os seguintes pacotes:



Configurar os módulos Java:

- Criar no pacote raiz “java” o arquivo “module.info.java”

```
// Responsável por definir as regras de visibilidade do modulo sistema.
module conta.sistema {
    // usa spring
    requires javax.inject;
    requires spring.tx;

    // expondo porta de entrada (driver)
    exports conta.sistema.casouso.porta;
    exports conta.sistema.casouso.imp;
    // expondo sistema negocio
    exports conta.sistema.dominio.modelo;
    exports conta.sistema.dominio.servico;
    // expondo adptadores de saidas (driven)
    exports conta.sistema.porta;
    exports conta.adaptador;

    // abre reflexão spring
    opens conta.sistema.casouso.porta;
    opens conta.sistema.casouso.imp;
    opens conta.sistema.dominio.servico;
    opens conta.adaptador;
}
```

Regras de Negócios

Criar a classe:

```
package conta.sistema.dominio.modelo;

// Responsável por representar um erro de negócio de sistema.
public class NegocioException extends RuntimeException {
    public NegocioException(String mensagem) {
        super(mensagem);
    }
}
```

Criar a classe:

```
package conta.sistema.dominio.modelo;

// Responsável por centralizar todas as mensagens de validações.
public class Erro {

    // eros genéricos
    public static void obrigatorio(String nome) {
        throw new NegocioException(nome + " é obrigatório.");
    }
    public static void inexistente(String nome) {
        throw new NegocioException(nome + " é inexistente.");
    }

    // eros especificos
    public static void saldoInsuficiente() {
        throw new NegocioException("Saldo insuficiente.");
    }
    public static void mesmaConta() {
        throw new NegocioException("Conta débito e crédito devem ser diferentes.");
    }
}
```

Criar classe:

```java
package conta.sistema.dominio.modelo;

import java.math.BigDecimal;

import static conta.sistema.dominio.modelo.Erro.obrigatorio;
import static conta.sistema.dominio.modelo.Erro.saldoInsuficiente;
import static java.util.Objects.isNull;

// Responsável por representar a entidade conta e suas regras.
// Não sera gerenciado pelo IoC e sim pelo repositorio.
public class Conta {
    private Integer numero;
    private BigDecimal saldo;
    private String correntista;

    public Conta() {
        numero = 0;
        saldo = BigDecimal.ZERO;
        correntista = "não informado";
    }

    public Conta(Integer numero, BigDecimal saldo, String correntista) {
        this.numero = numero;
        this.saldo = saldo;
        this.correntista = correntista;
    }

    public void creditar(BigDecimal credito) {
        if (isNull(credito)) {
            obrigatorio("Valor crédito");
        }
        if (credito.compareTo(BigDecimal.ZERO) <= 0) {
            obrigatorio("Valor crédito");
        }
        saldo = saldo.add(credito);
    }

    public void debitar(BigDecimal debito) {
        if (isNull(debito)) {
            obrigatorio("Valor débito");
        }
        if (debito.compareTo(BigDecimal.ZERO) <= 0) {
            obrigatorio("Valor débito");
        }
        if (debito.compareTo(saldo) > 0) {
            saldoInsuficiente();
        }
        saldo = saldo.subtract(debito);
    }

    // gets e sets

    public Integer getNumero() {
        return numero;
    }

    public void setNumero(Integer numero) {
        this.numero = numero;
    }
```

```
    public BigDecimal getSaldo() {
        return saldo;
    }

    public void setSaldo(BigDecimal saldo) {
        this.saldo = saldo;
    }

    public String getCorrentista() {
        return correntista;
    }

    public void setCorrentista(String correntista) {
        this.correntista = correntista;
    }

    @Override
    public String toString() {
        return "Conta{" +
                "numero=" + numero +
                ", saldo=" + saldo +
                ", correntista='" + correntista + '\'' +
                '}';
    }
}
```

Crie a classe de testes de unidade e execute:

```
package teste.unidade.dominio.modelo;

import conta.sistema.dominio.modelo.Conta;
import conta.sistema.dominio.modelo.NegocioException;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

import java.math.BigDecimal;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.fail;

@DisplayName("Regra de Crédito de Conta")
public class TesteCreditoConta {

    // armazena o saldo para teste ficar dinamico
    BigDecimal cem = new BigDecimal(100);
    Conta contaValida;

    @BeforeEach
    void prepara() {
        contaValida = new Conta(1, cem, "Rebeca");
    }

    // negativos

    @Test
    @DisplayName("valor crédito nulo como obrigatório")
    void teste1() {
        try {
            contaValida.creditar(null);
            fail("valor crédito obrigatório");
        } catch (NegocioException e) {
            assertEquals(e.getMessage(), "Valor crédito é obrigatório.");
```

```
            System.out.println(e.getMessage());
        }
    }

    @Test
    @DisplayName("valor crédito negativo como obrigatório")
    void teste2() {
        try {
            contaValida.creditar(new BigDecimal(-10));
            fail("valor crédito obrigatório");
        } catch (NegocioException e) {
            assertEquals(e.getMessage(), "Valor crédito é obrigatório.");
            System.out.println(e.getMessage());
        }
    }

    @Test
    @DisplayName("valor crédito zero como obrigatório")
    void teste3() {
        try {
            contaValida.creditar(BigDecimal.ZERO);
            fail("valor crédito obrigatório");
        } catch (NegocioException e) {
            assertEquals(e.getMessage(), "Valor crédito é obrigatório.");
            System.out.println(e.getMessage());
        }
    }

    // positivos

    @Test
    @DisplayName("valor crédito acima de zero")
    void teste4() {
        try {
            contaValida.creditar(BigDecimal.ONE);
            var saldoFinal = cem.add(BigDecimal.ONE);
            assertEquals(contaValida.getSaldo(), saldoFinal, "Saldo deve bater");
        } catch (NegocioException e) {
            fail("Deve creditar com sucesso - " + e.getMessage());
        }
    }
}
```

Crie a classe de testes de unidade e execute:

```
package teste.unidade.dominio.modelo;

import conta.sistema.dominio.modelo.Conta;
import conta.sistema.dominio.modelo.NegocioException;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

import java.math.BigDecimal;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.fail;

@DisplayName("Regra de Débito de Conta")
public class TesteDebitoConta {

    // armazena o saldo para teste ficar dinamico
    BigDecimal cem = new BigDecimal(1000);
    Conta contaValida;
```

```
    @BeforeEach
    void prepara() {
        contaValida = new Conta(1, cem, "Fernando");
    }

    // negativos

    @Test
    @DisplayName("valor débito nulo como obrigatório")
    void teste1() {
        try {
            contaValida.debitar(null);
            fail("valor débito é obrigatório");
        } catch (NegocioException e) {
            assertEquals(e.getMessage(), "Valor débito é obrigatório.");
            System.out.println(e.getMessage());
        }
    }

    @Test
    @DisplayName("valor débito negatico como obrigatório")
    void teste2() {
        try {
            contaValida.debitar(new BigDecimal(-10));
            fail("valor débito obrigatório");
        } catch (NegocioException e) {
            assertEquals(e.getMessage(), "Valor débito é obrigatório.");
            System.out.println(e.getMessage());
        }
    }

    @Test
    @DisplayName("valor débito zero como obrigatório")
    void teste3() {
        try {
            contaValida.debitar(BigDecimal.ZERO);
            fail("valor débito obrigatório");
        } catch (NegocioException e) {
            assertEquals(e.getMessage(), "Valor débito é obrigatório.");
            System.out.println(e.getMessage());
        }
    }

    @Test
    @DisplayName("valor débito acima do saldo")
    void teste4() {
        try {
            contaValida.debitar(cem.add(BigDecimal.ONE));
            fail("valor débito acima do saldo");
        } catch (NegocioException e) {
            assertEquals(e.getMessage(), "Saldo insuficiente.");
            System.out.println(e.getMessage());
        }
    }

    // positivos

    @Test
    @DisplayName("valor débito igual ao saldo")
    void teste5() {
        try {
            contaValida.debitar(cem);
            assertEquals(contaValida.getSaldo(), BigDecimal.ZERO, "Saldo deve zerar");
        } catch (NegocioException e) {
```

```
            fail("Deve debitar com sucesso - " + e.getMessage());
        }
    }

    @Test
    @DisplayName("valor débito menor que saldo")
    void teste6() {
        try {
            contaValida.debitar(BigDecimal.TEN);
            var saldoFinal = cem.subtract(BigDecimal.TEN);
            assertEquals(contaValida.getSaldo(), saldoFinal, "Saldo deve bater");
        } catch (NegocioException e) {
            fail("Deve debitar com sucesso - " + e.getMessage());
        }
    }
}
```

Processo de Negócios

Criar a classe:

```
package conta.sistema.dominio.servico;

import conta.sistema.dominio.modelo.Conta;

import javax.inject.Named;
import java.math.BigDecimal;

import static conta.sistema.dominio.modelo.Erro.obrigatorio;
import static java.util.Objects.isNull;

// Responsável por representar a entidade transferência e suas regras.
// Será gerenciado pelo IoC
@Named
public class Transferencia {

    public void processar(BigDecimal valor, Conta debito, Conta credito) {
        if (isNull(valor)) {
            obrigatorio("Valor da transferência");
        }
        if (isNull(debito)) {
            obrigatorio("Conta débito");
        }
        if (isNull(credito)) {
            obrigatorio("Conta crédito");
        }
        debito.debitar(valor);
        credito.creditar(valor);
    }
}
```

Crie a classe de testes de unidade e execute:

```
package teste.unidade.dominio.servico;

import conta.sistema.dominio.modelo.Conta;
import conta.sistema.dominio.modelo.NegocioException;
import conta.sistema.dominio.servico.Transferencia;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

import java.math.BigDecimal;
```

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.fail;

@DisplayName("Regra de Transferência")
public class TesteTransferencia {

    // armazena o cem para teste ficar dinamico
    BigDecimal cem = new BigDecimal(100);
    BigDecimal vinte = new BigDecimal(20);
    Transferencia trans = new Transferencia();
    Conta contaDebito;
    Conta contaCredito;

    @BeforeEach
    void prepara() {
        contaDebito = new Conta(1, cem, "Fernando");
        contaCredito = new Conta(2, cem, "Rebeca");
        trans = new Transferencia();
    }

    // negativos

    @Test
    @DisplayName("valor nulo como obrigatório")
    void teste1() {
        try {
            trans.processar(null, contaDebito, contaCredito);
            fail("valor transferência como obrigatório");
        } catch (NegocioException e) {
            assertEquals(e.getMessage(), "Valor da transferência é obrigatório.");
            System.out.println(e.getMessage());
        }
    }

    // Observação:
    // Não se faz necessário refazer os testes de nulo, zero ou negativo, na
transferência,
    // pois ele repassa para conta.debitar() e conta.creditar() os testes desses já
garantem isso.
    // Cada teste deve garantir o serviço implementadas dentro da classe a ser testada
e não
    // testar coisas de classes agregadas.

    @Test
    @DisplayName("conta debito como obrigatório")
    void teste2() {
        try {
            trans.processar(vinte, null, contaCredito);
            fail("conta debito obrigatório");
        } catch (NegocioException e) {
            assertEquals(e.getMessage(), "Conta débito é obrigatório.");
            System.out.println(e.getMessage());
        }
    }

    @Test
    @DisplayName("conta credito como obrigatório")
    void teste3() {
        try {
            trans.processar(vinte, contaDebito, null);
            fail("conta credito obrigatório");
        } catch (NegocioException e) {
            assertEquals(e.getMessage(), "Conta crédito é obrigatório.");
            System.out.println(e.getMessage());
```

```
        }
    }

    // positivos

    @Test
    @DisplayName("transferir 20 reais")
    void teste4() {
        try {
            trans.processar(vinte, contaDebito, contaCredito);
            assertEquals(contaDebito.getSaldo(), cem.subtract(vinte),
                    "Saldo da conta débito deve bater");
            assertEquals(contaCredito.getSaldo(), cem.add(vinte),
                    "Saldo da conta crédito deve bater");
        } catch (NegocioException e) {
            fail("Deve transferir com sucesso");
        }
    }
}
```

Porta dirigida de serviços externos: bancos de dados:

Criar interface:

```
package conta.sistema.porta;

import conta.sistema.dominio.modelo.Conta;
import conta.sistema.dominio.modelo.NegocioException;

// Responsável por definir a porta de saída (driven) de serviços de banco de dados.
public interface ContaRepositorio {

    Conta get(Integer numero);

    void alterar(Conta conta);
}
```

Criar o mock de serviços de bancos de dados:

```
package conta.adaptador;

import conta.sistema.dominio.modelo.Conta;
import conta.sistema.dominio.modelo.NegocioException;
import conta.sistema.porta.ContaRepositorio;

import javax.inject.Named;
import java.math.BigDecimal;
import java.util.HashMap;
import java.util.Map;

import static java.util.Objects.isNull;

// Responsável por implementar a porta de saída (driven) de serviços de banco de dados
falso.
// será gerenciado pelo IoC
@Named
public class AdaptadorContaFakeImp implements ContaRepositorio {

    private Map<Integer, Conta> banco = new HashMap<>();
```

```
    public AdaptadorContaFakeImp() {
        banco.put(10, new Conta(10, new BigDecimal(100), "Fernando Fake"));
        banco.put(20, new Conta(20, new BigDecimal(100), "Rebeca Fake"));
    }

    public Conta get(Integer numero) {
        System.out.println("Fake banco de dados -> Conta get(numero)");
        return banco.get(numero);
    }

    public void alterar(Conta conta) {
        System.out.println("Fake banco de dados -> alterar(conta)");
        var ct = banco.get(conta.getNumero());
        if (!isNull(ct)) {
            banco.put(conta.getNumero(), conta);
        } else {
            throw new NegocioException("Conta inexistente: " + conta.getNumero());
        }
    }
}
```

Porta condutora de caso de uso:

Criar interface:

```
package conta.sistema.casouso.porta;

import conta.sistema.dominio.modelo.Conta;
import conta.sistema.dominio.modelo.NegocioException;

import java.math.BigDecimal;

// Responsável por definir a porta de entrada (driver) de operações para caso de uso de
transferência.
public interface PortaTransferencia {
    Conta getConta(Integer numero);
    void transferir(Integer contaDebito, Integer contaCredito, BigDecimal valor);
}
```

Criar classe controladora de caso de uso:

```
package conta.sistema.casouso.imp;

import conta.sistema.casouso.porta.PortaTransferencia;
import conta.sistema.dominio.modelo.Conta;
import conta.sistema.dominio.servico.Transferencia;
import conta.sistema.porta.ContaRepositorio;
import org.springframework.transaction.annotation.Transactional;

import javax.inject.Inject;
import javax.inject.Named;
import java.math.BigDecimal;

import static conta.sistema.dominio.modelo.Erro.*;
import static java.util.Objects.isNull;

// Responsável por implementar a porta de operações para caso de uso de transferência.
// Sera gerenciado pelo IoC
@Named
public class PortaTransferenciaImp implements PortaTransferencia {

    private ContaRepositorio repositorio;
```

```
    private Transferencia transferencia;

    // Ioc por construtor
    @Inject
    public PortaTransferenciaImp(ContaRepositorio repositorio, Transferencia
transferencia) {
        this.repositorio = repositorio;
        this.transferencia = transferencia;
    }

    @Override
    public Conta getConta(Integer numero) {
        return repositorio.get(numero);
    }

    @Override
    @Transactional
    public void transferir(Integer contaDebito, Integer contaCredito, BigDecimal valor)
{
        //1. validação de parametros
        if (isNull(contaDebito)) {
            obrigatorio("Conta débito");
        }
        if (isNull(contaCredito)) {
            obrigatorio("Conta crédito");
        }
        if (isNull(valor)) {
            obrigatorio("Valor");
        }

        //2. validação de contas
        var debito = repositorio.get(contaDebito);
        if (isNull(debito)) {
            inexistente("Conta débito");
        }
        var credito = repositorio.get(contaCredito);
        if (isNull(credito)) {
            inexistente("Conta crédito");
        }

        //3.validacao mesma conta
        if (debito.getNumero().equals(credito.getNumero())) {
            mesmaConta();
        }

        //4. operação
        transferencia.processar(valor, debito, credito);
        repositorio.alterar(debito);
        repositorio.alterar(credito);
    }
}
```

Crie a configuração IoC Spring de build 1 – “Sistema <- Adaptador Mocks”.

```
package teste.casouso;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

// Responsável por configurar os serviços do spring
@Configuration
@ComponentScan({
        // objetos de sistema
        "conta.sistema",
        // adptadores falsos
        "conta.adaptador"})
public class Build1 {
    // Build 1: Adaptador Testes -> Sistema <- Adptadores Mocks
}
```

Crie a classe de testes de aceitação e execute:

```
package teste.casouso;

import conta.sistema.casouso.porta.PortaTransferencia;
import conta.sistema.dominio.modelo.NegocioException;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit.jupiter.SpringExtension;

import javax.inject.Inject;
import java.math.BigDecimal;

import static org.junit.jupiter.api.Assertions.*;

@DisplayName("Caso de Uso - Serviço de Transferência")
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = Build1.class)
public class TesteAdaptadorTransferencia {

    Integer contaCredito = 10;
    Integer contaDebito = 20;
    Integer contaInexistente = 30;
    BigDecimal cem = new BigDecimal(100);
    BigDecimal cinquenta = new BigDecimal(50);

    @Inject
    PortaTransferencia porta;

    // negativos get conta

    @Test
    @DisplayName("pesquisa conta com número nulo")
    void teste1() {
        try {
            var conta = porta.getConta(null);
            assertTrue(conta == null, "Conta deve ser nula");
        } catch (NegocioException e) {
            fail("Deva carregar uma conta nula.");
        }
    }
```

```java
    @Test
    @DisplayName("pesquisa conta com número inexistente")
    void teste2() {
        try {
            var conta = porta.getConta(contaInexistente);
            assertTrue(conta == null, "Conta deve ser nula");
        } catch (NegocioException e) {
            fail("Deva carregar uma conta nula.");
        }
    }

    // positivo get conta

    @Test
    @DisplayName("pesquisa conta com número existente")
    void teste3() {
        try {
            var conta = porta.getConta(contaCredito);
            assertTrue(conta != null, "Conta deve estar preenchida");
            System.out.println(conta);
        } catch (NegocioException e) {
            fail("Deva carregar uma conta existente.");
        }
    }


    // negativos transferencia

    @Test
    @DisplayName("conta crédito como obrigatório")
    void teste4() {
        try {
            porta.transferir(null, contaCredito, cinquenta);
            fail("Conta débito é obrigatório");
        } catch (NegocioException e) {
            assertEquals(e.getMessage(), "Conta débito é obrigatório.");
            System.out.println(e.getMessage());
        }
    }

    @Test
    @DisplayName("conta débito como obrigatório")
    void teste5() {
        try {
            porta.transferir(contaDebito, null, cinquenta);
            fail("Conta crédito é obrigatório");
        } catch (NegocioException e) {
            assertEquals(e.getMessage(), "Conta crédito é obrigatório.");
            System.out.println(e.getMessage());
        }
    }

    @Test
    @DisplayName("valor como obrigatório")
    void teste6() {
        try {
            porta.transferir(contaDebito, contaCredito, null);
            fail("Valor é obrigatório");
        } catch (NegocioException e) {
            assertEquals(e.getMessage(), "Valor é obrigatório.");
            System.out.println(e.getMessage());
        }
    }
```

```
    @Test
    @DisplayName("conta débito inexistente")
    void teste7() {
        try {
            porta.transferir(contaInexistente, contaCredito, cinquenta);
            fail("Conta débito inexistente");
        } catch (NegocioException e) {
            assertEquals(e.getMessage(), "Conta débito inexistente.");
            System.out.println(e.getMessage());
        }
    }

    @Test
    @DisplayName("conta crédito inexistente")
    void teste8() {
        try {
            porta.transferir(contaDebito, contaInexistente, cinquenta);
            fail("Conta crédito inexistente");
        } catch (NegocioException e) {
            assertEquals(e.getMessage(), "Conta crédito inexistente.");
            System.out.println(e.getMessage());
        }
    }

    @Test
    @DisplayName("mesma conta débito e crédito")
    void teste9() {
        try {
            porta.transferir(contaDebito, contaDebito, cinquenta);
            fail("Conta crédito e debito deve ser diferentes");
        } catch (NegocioException e) {
            assertEquals(e.getMessage(), "Conta débito e crédito devem ser
diferentes.");
            System.out.println(e.getMessage());
        }
    }

    // Observação:
    // Não se faz necessário refazer os testes de valor nulo, zero ou negativo, no caso
de uso, pois ele repassa os
    // objetos internos de domínio já testados. Cada teste deve garantir o serviço
implementadas dentro da classe a
    // ser testada e não testar coisas de classes agregadas.

    // positivo transferência

    @Test
    @DisplayName("transferência de 50 reais")
    void teste10() {
        try {
            porta.transferir(contaDebito, contaCredito, cinquenta);
        } catch (NegocioException e) {
            fail("Não deve gerar erro de transferência - " + e.getMessage());
        }
        try {
            var credito = porta.getConta(contaCredito);
            var debito = porta.getConta(contaDebito);
            assertEquals(credito.getSaldo(), cem.add(cinquenta), "Saldo crédito deve
bater");
            assertEquals(debito.getSaldo(), cem.subtract(cinquenta), "Saldo débito deve
bater");
        } catch (NegocioException e) {
            fail("Não deve gerar erro de validação de saldo - " + e.getMessage());
        }
```

```
    }
}
```

Crie a classe de suíte de teste geral e execute:

```
package teste;

import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectPackages({
        // testando regras
        "teste.unidade.dominio.modelo",
        // testando servicos
        "teste.unidade.dominio.servico",
        // testando porta entrada (driver)
        "teste.casouso"})
public class SuiteCore {
    // 100% da solução testada independente de front-end e serviços externos (banco de
dados)
}
```

- Total de 24 testes = Core do sistema 100% testado e funcional com TDD.

Gerar um jar chamado “sistema.jar” que será reusado nos outros projetos:

- Menu: File-> Project Structure:
- Opção: Artifacts -> + (adicionar) Jar -> Empty.
- Colocar o “unnamed” como “sistema”.
- Setar nome de “unnamed.jar” por “sistema.jar”
- Adicionar byte codes dentro do jar “compile output”.
- Adicionar fontes dentro do jar (+) “Module Sources”
- Criar um manifest vazio.
- Clicar no botão “Apply” e “Ok”.

Para gerar o jar siga o menu: Build->Build Artifacts->sistema->build.

PROJETO FRONT-END JAVAFX

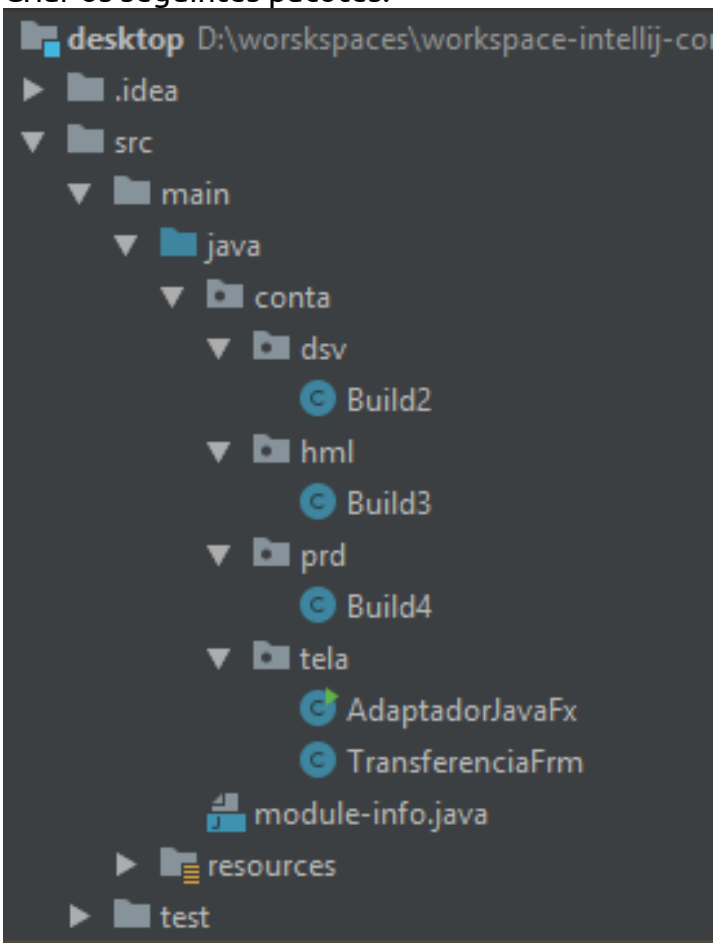
Criar um projeto do **lado esquerdo** “Adaptadores Primários”:

- Crie projeto chamado Maven “desktop”.
- Configure as dependências no pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>desktop</groupId>
    <artifactId>desktop</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>12</maven.compiler.source>
        <maven.compiler.target>12</maven.compiler.target>
    </properties>

    <dependencies>

        <!-- Conta Sistema -->
        <dependency>
            <groupId>conta.sistema</groupId>
            <artifactId>conta.sistema</artifactId>
            <version>1.0.0</version>
            <scope>system</scope>
            <systemPath>D:/worskspaces/workspace-intellij-
comunity/sistema/out/artifacts/sistema/sistema.jar
            </systemPath>
        </dependency>

        <!-- JavaFx -->
        <dependency>
            <groupId>org.openjfx</groupId>
            <artifactId>javafx-controls</artifactId>
            <version>12</version>
        </dependency>

        <!-- CDI -->
        <dependency>
            <groupId>javax.enterprise</groupId>
            <artifactId>cdi-api</artifactId>
            <version>2.0</version>
        </dependency>

        <!-- Spring IoC -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.6.RELEASE</version>
        </dependency>

        <!-- Spring Transactions -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-tx</artifactId>
            <version>5.1.6.RELEASE</version>
        </dependency>
```

```
    </dependencies>
</project>
```

Criar os seguintes pacotes:



Configurar os módulos Java

- Criar no pacote raiz “java” o arquivo “module.info.java”:

```
module conta.desktop {
    // usa conta sistema
    requires conta.sistema;
    // usa spring
    requires javax.inject;
    requires spring.tx;
    requires spring.core;
    requires spring.beans;
    requires spring.context;
    requires java.sql;
    // usa javafx
    requires javafx.controls;

    // abre telas e builds
    opens conta.tela;
    opens conta.dsv;
    opens conta.hml;
    opens conta.prd;
}
```

Criar a classe de build 2:

```
package conta.dsv;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

// Responsável por configurar os serviços do spring
@Configuration
@ComponentScan({
        // adptadores front-end javafx
        "conta.dsv",
        "conta.tela",
        // core do sistema
        "conta.sistema",
        // adptadores falsos
        "conta.adaptador"})
public class Build2 {
    // Build 2 - Adaptador JavaFX -> Sistema <- Adaptadores Mocks
}
```

Criar classe adaptador primário JavaFX:

```
package conta.tela;

import conta.sistema.casouso.porta.PortaTransferencia;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;

import javax.inject.Inject;
import javax.inject.Named;
import java.math.BigDecimal;

import static java.util.Objects.isNull;

// Responsável por desenhar a tela de transferência com a tecnologia javafx.
@Named
public class TransferenciaFrm {

    private TextField tfDebito;
    private TextField tfNomeDebito;
    private TextField tfCredito;
    private TextField tfNomeCredito;
    private TextField tfValor;
    private PortaTransferencia porta;

    @Inject
    public TransferenciaFrm(PortaTransferencia porta) {
        this.porta = porta;
    }

    private void limpar() {
        tfDebito.setText("");
        tfNomeDebito.setText("");
        tfCredito.setText("");
        tfNomeCredito.setText("");
        tfValor.setText("");
    }
```

```
    private Integer get(String valor) {
        try {
            return Integer.valueOf(valor);
        } catch (Exception e) {
            return null;
        }
    }

    private void mensagem(String texto) {
        var alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setTitle("Transferência Bancária");
        alert.setHeaderText(null);
        alert.setContentText(texto);
        alert.showAndWait();
    }

    private void get(TextField tfEntrada, TextField tfSaida) {
        try {
            var conta = porta.getConta(get(tfEntrada.getText()));
            if (isNull(conta)) {
                tfSaida.setText("");
            } else {
                tfSaida.setText(conta.getCorrentista() + " - Saldo R$ " +
conta.getSaldo());
            }
        } catch (Exception e) {
            mensagem(e.getMessage());
        }
    }

    private BigDecimal get() {
        try {
            return new BigDecimal(tfValor.getText());
        } catch (Exception e) {
            return null;
        }
    }


    private FlowPane montarTela() {
        var pn = new FlowPane();
        pn.setHgap(10);
        pn.setVgap(10);

        pn.getChildren().add(new Label(" Conta Débito:"));
        tfDebito = new TextField();
        tfDebito.setPrefWidth(50);
        tfDebito.focusedProperty().addListener((o, v, n) -> {
            if (!n) get(tfDebito, tfNomeDebito);
        });

        pn.getChildren().add(tfDebito);

        tfNomeDebito = new TextField();
        tfNomeDebito.setPrefWidth(300);
        tfNomeDebito.setEditable(false);
        pn.getChildren().add(tfNomeDebito);

        pn.getChildren().add(new Label(" Conta Crédito:"));
        tfCredito = new TextField();
        tfCredito.setPrefWidth(50);
        tfCredito.focusedProperty().addListener((o, v, n) -> {
            if (!n) get(tfCredito, tfNomeCredito);
        });
        pn.getChildren().add(tfCredito);
```

```
        tfNomeCredito = new TextField();
        tfNomeCredito.setEditable(false);
        tfNomeCredito.setPrefWidth(300);
        pn.getChildren().add(tfNomeCredito);

        pn.getChildren().add(new Label(" Valor R$:"));
        tfValor = new TextField();
        tfValor.setPrefWidth(200);
        pn.getChildren().add(tfValor);

        var bt = new Button();
        bt.setOnAction((ev) -> {
            try {
                porta.transferir(get(tfDebito.getText()), get(tfCredito.getText()),
get());
                limpar();
                mensagem("Transferência feita com sucesso!");
            } catch (Exception e) {
                mensagem(e.getMessage());
            }
        });
        bt.setText("Transferir");
        pn.getChildren().add(bt);
        return pn;
    }

    public void mostrar(Stage stage) {
        stage.setTitle("Adaptador JavaFX");
        var scene = new Scene(montarTela(), 500, 100);
        stage.setScene(scene);
        stage.show();
    }
}
```

Criar a classe início desktop:

```
package conta.tela;

import conta.dsv.Build2;
import javafx.application.Application;
import javafx.stage.Stage;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

// Responsável por fazer o ponto de inicio de execução
public class AdaptadorJavaFx extends Application {

    private ApplicationContext spring;

    @Override
    public void init() {
        System.out.println("iniciando spring..");
        spring = new AnnotationConfigApplicationContext(Build2.class);
    }

    @Override
    public void start(Stage stage) {
        var form = spring.getBean(TransferenciaFrm.class);
        form.mostrar(stage);
    }

    public static void main(String[] args) { launch(args);}
}
```

- Execute o AdaptadorJavaFx e faça o teste usando “adaptador secundário” simulado mock.

PROJETO BACKSERVICES

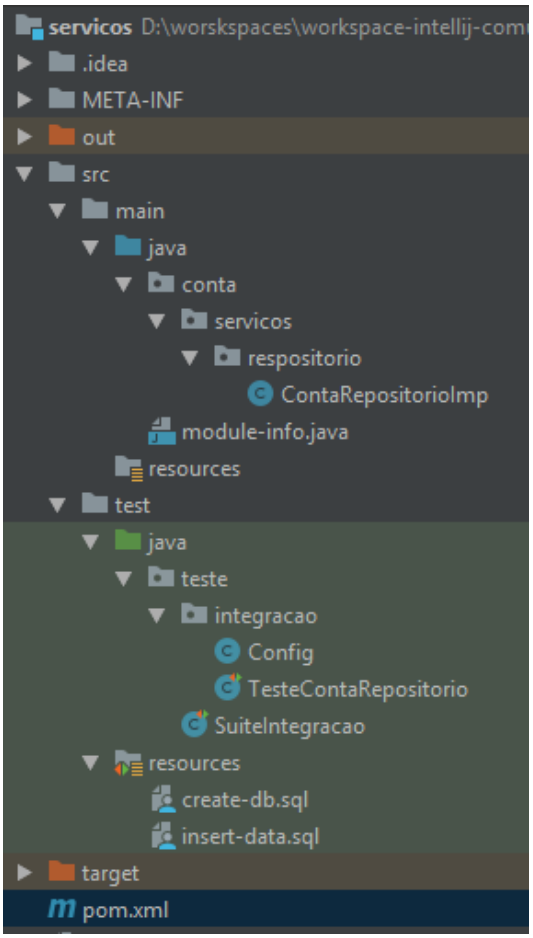
Criar um projeto do **lado direito** “Adaptadores Secundários”:

- Crie projeto chamado Maven “servicos”.
- Configure as dependências no pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>servicos</groupId>
    <artifactId>servicos</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>12</maven.compiler.source>
        <maven.compiler.target>12</maven.compiler.target>
    </properties>

    <dependencies>

        <!-- JUnit -->
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>5.4.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.platform</groupId>
            <artifactId>junit-platform-runner</artifactId>
            <version>1.4.1</version>
            <scope>test</scope>
        </dependency>

        <!-- Conta Sistema -->
        <dependency>
            <groupId>sistema.sistema</groupId>
            <artifactId>sistema.sistema</artifactId>
            <version>1.0.0</version>
            <scope>system</scope>
            <systemPath>D:/worskspaces/workspace-intellij-
comunity/sistema/out/artifacts/sistema/sistema.jar
            </systemPath>
        </dependency>

        <!-- CDI -->
        <dependency>
            <groupId>javax.enterprise</groupId>
            <artifactId>cdi-api</artifactId>
            <version>2.0</version>
        </dependency>

        <!-- Spring IoC -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.6.RELEASE</version>
        </dependency>

        <!-- Spring Transactions -->
```

```
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-tx</artifactId>
            <version>5.1.6.RELEASE</version>
        </dependency>

        <!-- Spring Jdbc -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>5.1.6.RELEASE</version>
        </dependency>

        <!-- Spring Test -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>5.1.6.RELEASE</version>
            <scope>test</scope>
        </dependency>

        <!-- hsqldb -->
        <dependency>
            <groupId>org.hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>2.4.1</version>
            <scope>test</scope>
        </dependency>

    </dependencies>
</project>
```

Criar os seguintes pacotes:



Adicione na pasta “resources” de test os arquivos:

- create-db.sql
- insert-db.sql

Configurar os módulos Java:

- Criar no pacote raiz “java” o arquivo “module.info.java”

```
module conta.servicos {
    // usa sistema
    requires conta.sistema;

    // usa spring
    requires javax.inject;
    requires spring.tx;
    requires spring.core;
    requires spring.beans;
    requires spring.context;
    requires java.sql;
    requires spring.jdbc;

    // abre respositorio
    opens conta.servicos.respositorio;
}
```

Crie a classe:

```
package conta.servicos.respositorio;

import conta.sistema.dominio.modelo.Conta;
import conta.sistema.dominio.modelo.NegocioException;
import conta.sistema.porta.ContaRepositorio;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.transaction.annotation.Transactional;

import javax.inject.Inject;
import javax.inject.Named;

import static java.util.Objects.isNull;

// Responsável por implementar a porta de saída (driven) de serviços de banco de dados
usando spring jdbc
@Named
public class ContaRepositorioImp implements ContaRepositorio {

    private static final String ERRO = "Erro inesperado de acesso ao banco. Entre em
contato com adminstrador.";
    private JdbcTemplate jdbc;

    @Inject
    public ContaRepositorioImp(JdbcTemplate jdbcTemplate) {
        this.jdbc = jdbcTemplate;
    }

    @Override
    public Conta get(Integer numero) {
        if (isNull(numero)) {
            return null;
        }
        var sql = "select * from conta where numero = ?";
        var pm = new Object[]{numero};
        RowMapper<Conta> orm = (rs, nm) ->
                new Conta(rs.getInt(1), rs.getBigDecimal(2), rs.getString(3));
```

```
        try {
            var lista = jdbc.query(sql, pm, orm);
            if (lista.isEmpty()) {
                return null;
            }
            return lista.get(0);
        } catch (Exception e) {
            e.printStackTrace();
            throw new NegocioException(ERRO);
        }
    }

    @Transactional
    @Override
    public void alterar(Conta conta) {
        if (isNull(conta)) {
            throw new NegocioException("Conta é obrigatório.");
        }
        var sql = "update conta set saldo=?, correntista=? where numero=?";
        var pm = new Object[]{conta.getSaldo(), conta.getCorrentista(),
conta.getNumero()};
        try {
            jdbc.update(sql, pm);
        } catch (Exception e) {
            e.printStackTrace();
            throw new NegocioException(ERRO);
        }
    }
}
```

Crie a classe:

```
package teste.integracao;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.sql.DataSource;

// Responsável por configurar os serviços do spring
@Configuration
@EnableTransactionManagement
@ComponentScan({"conta.servicos.respositorio"})
public class Config {

    @Bean
    public DataSource dataSource() {
        var builder = new EmbeddedDatabaseBuilder();
        var db = builder.setType(EmbeddedDatabaseType.HSQL.HSQL)
                .addScript("create-db.sql")
                .addScript("insert-data.sql")
                .build();
        return db;
    }
```

```
    @Bean
    public JdbcTemplate jdbcTemplate() {
        return new JdbcTemplate(dataSource());
    }

    @Bean
    public DataSourceTransactionManager txManager() {
        return new DataSourceTransactionManager(dataSource());
    }
}
```

Crie a classe de teste de integração e execute:

```
package teste.integracao;

import conta.sistema.dominio.modelo.NegocioException;
import conta.sistema.porta.ContaRepositorio;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit.jupiter.SpringExtension;

import javax.inject.Inject;
import java.math.BigDecimal;

import static org.junit.jupiter.api.Assertions.*;

@DisplayName("Serviço de banco de dados - Conta")
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = Config.class)
public class TesteContaRepositorio {

    @Inject
    ContaRepositorio rep;

    // negativos

    @Test
    @DisplayName("pesquisa conta com número nulo")
    void teste1() {
        try {
            var conta = rep.get(null);
            assertTrue(conta == null, "Conta deve ser nula");
        } catch (NegocioException e) {
            fail("Deve carregar uma conta nula.");
        }
    }

    @Test
    @DisplayName("pesquisa conta com número inexistente")
    void teste2() {
        try {
            var conta = rep.get(8547);
            assertTrue(conta == null, "Conta deve ser nula");
        } catch (NegocioException e) {
            fail("Deve carregar uma conta nula.");
        }
    }

    // positivo

    @Test
    @DisplayName("pesquisa conta com número existente")
```

```
    void teste3() {
        try {
            var conta = rep.get(50);
            assertTrue(conta != null, "Conta deve estar preenchida");
            System.out.println(conta);
        } catch (NegocioException e) {
            fail("Deve carregar uma conta.");
        }
    }

    // negativos

    @Test
    @DisplayName("alterar conta como nulo")
    void teste4() {
        try {
            rep.alterar(null);
            fail("Não deve alterar conta nula");
        } catch (NegocioException e) {
            assertEquals(e.getMessage(), "Conta é obrigatório.");
            System.out.println(e.getMessage());
        }
    }

    // positivo

    @Test
    @DisplayName("alterar conta com sucesso")
    void teste5() {
        try {
            var c1 = rep.get(50);
            c1.setSaldo(new BigDecimal("1.00"));
            c1.setCorrentista("Alterado");
            rep.alterar(c1);

            var c2 = rep.get(50);
            assertEquals(c1.getSaldo(), c2.getSaldo(), "Deve bater o saldo");
            assertEquals(c1.getCorrentista(), c2.getCorrentista(), "Deve bater o
correntista");
        } catch (NegocioException e) {
            fail("Deve alterar conta ");
        }
    }
}
```

Criar a classe de suíte de integração e execute:

```
package teste;

import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectPackages({"teste.integracao"})
public class SuiteIntegracao {
    // 100% da integração com serviços externos testados.
}
```

- Total de 5 testes = adaptador com banco de dados 100% testado e funcional com TDD.

Gerar um jar chamado “servicos.jar” que será reusado nos outros projetos:

- Menu: File-> Project Structure:

- Opção: Artifacts -> + (adicionar) Jar -> Empty.
- Colocar o “unnamed” como “servicos”.
- Setar nome de “unnamed.jar” por “servicos.jar”
- Adicionar byte codes dentro do jar “compile output”.
- Adicionar fontes dentro do jar (+) “Module Sources”
- Criar um manifest vazio.
- Clicar no botão “Apply” e “Ok”.

Para gerar o jar siga o menu: Build->Build Artifacts-> servicos ->build.

PROJETO FRONT-END JAVAFX

Alterar o projeto fazendo o “build 3” homologação:

Adaptador JavaFX -> Sistema <- Adaptadores Real em Homologação

Altere o pom.xml com o serviços.jar e os frameworks de persistências.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>desktop</groupId>
    <artifactId>desktop</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>12</maven.compiler.source>
        <maven.compiler.target>12</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- Conta Sistema -->
        <dependency>
            <groupId>conta.sistema</groupId>
            <artifactId>conta.sistema</artifactId>
            <version>1.0.0</version>
            <scope>system</scope>
            <systemPath>D:/worskspaces/workspace-intellij-
comunity/sistema/out/artifacts/sistema/sistema.jar
            </systemPath>
        </dependency>

        <!-- Conta Servicos -->
        <dependency>
            <groupId>conta.servicos</groupId>
            <artifactId>conta.servicos</artifactId>
            <version>1.0.0</version>
            <scope>system</scope>
            <systemPath>D:/worskspaces/workspace-intellij-
comunity/servicos/out/artifacts/servicos/servicos.jar
            </systemPath>
        </dependency>

        <!-- JavaFx -->
        <dependency>
            <groupId>org.openjfx</groupId>
            <artifactId>javafx-controls</artifactId>
            <version>12</version>
        </dependency>

        <!-- CDI -->
        <dependency>
            <groupId>javax.enterprise</groupId>
            <artifactId>cdi-api</artifactId>
            <version>2.0</version>
        </dependency>

        <!-- Spring IoC -->
        <dependency>
            <groupId>org.springframework</groupId>
```

```
            <artifactId>spring-context</artifactId>
            <version>5.1.6.RELEASE</version>
        </dependency>

        <!-- Spring Transactions -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-tx</artifactId>
            <version>5.1.6.RELEASE</version>
        </dependency>

        <!-- Spring Jdbc -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>5.1.6.RELEASE</version>
        </dependency>

        <!-- hsqldb -->
        <dependency>
            <groupId>org.hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>2.4.1</version>
        </dependency>
    </dependencies>
</project>
```

Alterar o modulo-info.java habilitando os novos módulos, adaptador primário de produção e serviços de persistências:

```
module conta.desktop {
    // ==> BUILD 2
    // usa conta sistema
    requires conta.sistema;
    // usa spring
    requires javax.inject;
    requires spring.tx;
    requires spring.core;
    requires spring.beans;
    requires spring.context;
    requires java.sql;
    // usa javafx
    requires javafx.controls;

    // abre telas e builds
    opens conta.tela;
    opens conta.dsv;
    opens conta.hml;
    opens conta.prd;

    // ==> BUILD 3 e 4
    // usa conta serviços
    requires conta.servicos;
    // usa spring jdbc da conta serviços
    requires spring.jdbc;
    requires hsqldb;
}
```

Criar a classe de build 3:

```
package conta.hml;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.sql.DataSource;

// Responsável por configurar os serviços do spring
@Configuration
@EnableTransactionManagement
@ComponentScan({
        // adptadores front-end javafx
        "conta.tela",
        "conta.hml",
        // core do sistema
        "conta.sistema",
        // adptadores real
        "conta.servicos.respositorio"})
public class Build3 {

    // Build 3 - Adaptador JavaFX -> Sistema <- Adaptadores Real em Homologação

    @Bean
    public DataSource dataSource() {
        var builder = new EmbeddedDatabaseBuilder();
        var db = builder.setType(EmbeddedDatabaseType.HSQL.HSQL)
                .addScript("create-db.sql")
                .addScript("insert-hml.sql")
                .build();
        return db;
    }

    @Bean
    public JdbcTemplate jdbcTemplate() {
        return new JdbcTemplate(dataSource());
    }

    @Bean
    public DataSourceTransactionManager txManager() {
        return new DataSourceTransactionManager(dataSource());
    }
}
```

Adicione os arquivos sql no resouces do projeto:

- create-db.sql
- insert-hml.sql
- insert-prd.sql

Alterar a classe para executar o build 3:

```
package conta.tela;

import conta.dsv.Build2;
import conta.hml.Build3;
import javafx.application.Application;
import javafx.stage.Stage;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

// Responsável por fazer o ponto de inicio de execução
public class AdaptadorJavaFx extends Application {

    private ApplicationContext spring;

    @Override
    public void init() {
        System.out.println("iniciando spring..");
        //spring = new AnnotationConfigApplicationContext(Build2.class);
        spring = new AnnotationConfigApplicationContext(Build3.class);
    }

    @Override
    public void start(Stage stage) {
        var form = spring.getBean(TransferenciaFrm.class);
        form.mostrar(stage);
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

- Execute o AdaptadorJavaFx e faça o teste usando adaptador secundário real, usando base de homologação.

PROJETO FRONT-END JAVAFX

Alterar o projeto fazendo o “build 4” produção:

Adaptador JavaFX -> Sistema <- Adaptadores Real em Produção

Crie o banco de dados HSQLDB em produção:

- Veja instalação-hsqldb.pdf.

Crie a classe de build 4:

```
package conta.prd;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.jdbc.datasource.SimpleDriverDataSource;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.sql.DataSource;

// Responsável por configurar os serviços do spring
@Configuration
@EnableTransactionManagement
@ComponentScan({
        // adptadores front-end javafx
        "conta.tela",
        "conta.prd",
        // core do sistema
        "conta.sistema",
        // adptadores hsqdl
        "conta.servicos.respositorio"})
public class Build4 {

    //Build 4 - Adaptador JavaFX -> Sistema <- Adaptadores Real em Produção

    @Bean
    public DataSource dataSource() {
        var ds = new SimpleDriverDataSource();
        ds.setDriverClass(org.hsqldb.jdbcDriver.class);
        ds.setUrl("jdbc:hsqldb:file:D:/Java/hsql-2.4.1/conta/");
        ds.setUsername("sa");
        ds.setPassword("1234");
        return ds;
    }

    @Bean
    public JdbcTemplate jdbcTemplate() {
        return new JdbcTemplate(dataSource());
    }

    @Bean
    public DataSourceTransactionManager txManager() {
        return new DataSourceTransactionManager(dataSource());
    }
}
```

Alterar a classe para executar o build 4:

```
package conta.tela;

import conta.prd.Build4;
import javafx.application.Application;
import javafx.stage.Stage;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

// Responsável por fazer o ponto de inicio de execução
public class AdaptadorJavaFx extends Application {

    private ApplicationContext spring;

    @Override
    public void init() {
        System.out.println("iniciando spring..");
        //spring = new AnnotationConfigApplicationContext(Build2.class);
        //spring = new AnnotationConfigApplicationContext(Build3.class);
        spring = new AnnotationConfigApplicationContext(Build4.class);
    }

    @Override
    public void start(Stage stage) {
        var form = spring.getBean(TransferenciaFrm.class);
        form.mostrar(stage);
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

- Execute o AdaptadorJavaFx e faça o teste usando adaptador secundário real, usando base de produção.