CURSO DE TESTES AUTOMATIZADOS

DBUNIT

Wagner Costa

wcaquino@gmail.com

TESTES DEPENDENTES

- Cenário dos testes:
  - Teste 1: Insere um registro
  - Teste 2: Detalha os dados do registro (T1)
  - Teste 3: Altera dos dados do registro (T1)
  - Teste 4: Excluir o registro (T3)
- O que acontece se o Teste 1 não for realizado com sucesso?
- Qual o procedimento caso queira testar apenas o teste 4?

MASSA DE DADOS CONHECIDA

- O uso de uma massa de dados conhecida é primordial para este tipo de operação.
  - Poderiamos criar um script SQL para executá-lo sempre antes de executar cada um dos testes a todas as horas durante todo o tempo do projeto.
    - Vai ficar chato!
    - Adeus automatização!

DBUNIT

- É um framework opensource criado por Manuel Laflamme.
- É uma ferramenta poderosa que simplifica a utilização de operações com bancos de dados nos testes unitários
- Versão atual: 2.4.9;

PRA QUE SERVE DBUNIT?

- Garante que a base de dados estará sempre num estado conhecido
- É uma excelente forma de se livrar dos problemas causados por testes que corrompem a base, fazendo com que os testes seguintes venham a falhar

VANTAGENS

- Simplifica a preparação dos dados para os testes
- Provê um mecanismo simples (baseado em XML) para carregar os dados
- Provê um mecanismo igualmente simples (baseado em XML) para exportar os dados
- Pode trabalhar com bases de dados grandes
- Ajuda a verificar se o estado atual do banco está de acordo com o conjunto de dados esperado

INSTALAÇÃO

- Para instalar o DBUnit, basta adicionar a lib dbunit-<versão>.jar no classpath do projeto
  - Porém o DBUnit possui dependências:
    - JUnit
  - SLF4J
- Também deve-se adicionar o driver do banco de dados que será utilizado

A CLASSE DE TESTES

- Deve estender a classe DatabaseTestCase
- Os métodos *getConnection* e *getDataSet* devem ser implementados
- Os métodos **devem** iniciar com test
  - De volta aos padrões do JUnit 3.x
  - Adeus Annotations?

GETCONNECTION

- É um método abstrato da classe DatabaseTestCase
- Deve retornar um objeto que implemente a interface IDatabaseConnection
- É responsável por retornar a conexão com a base de dados para todas as operações com o banco dentro do teste

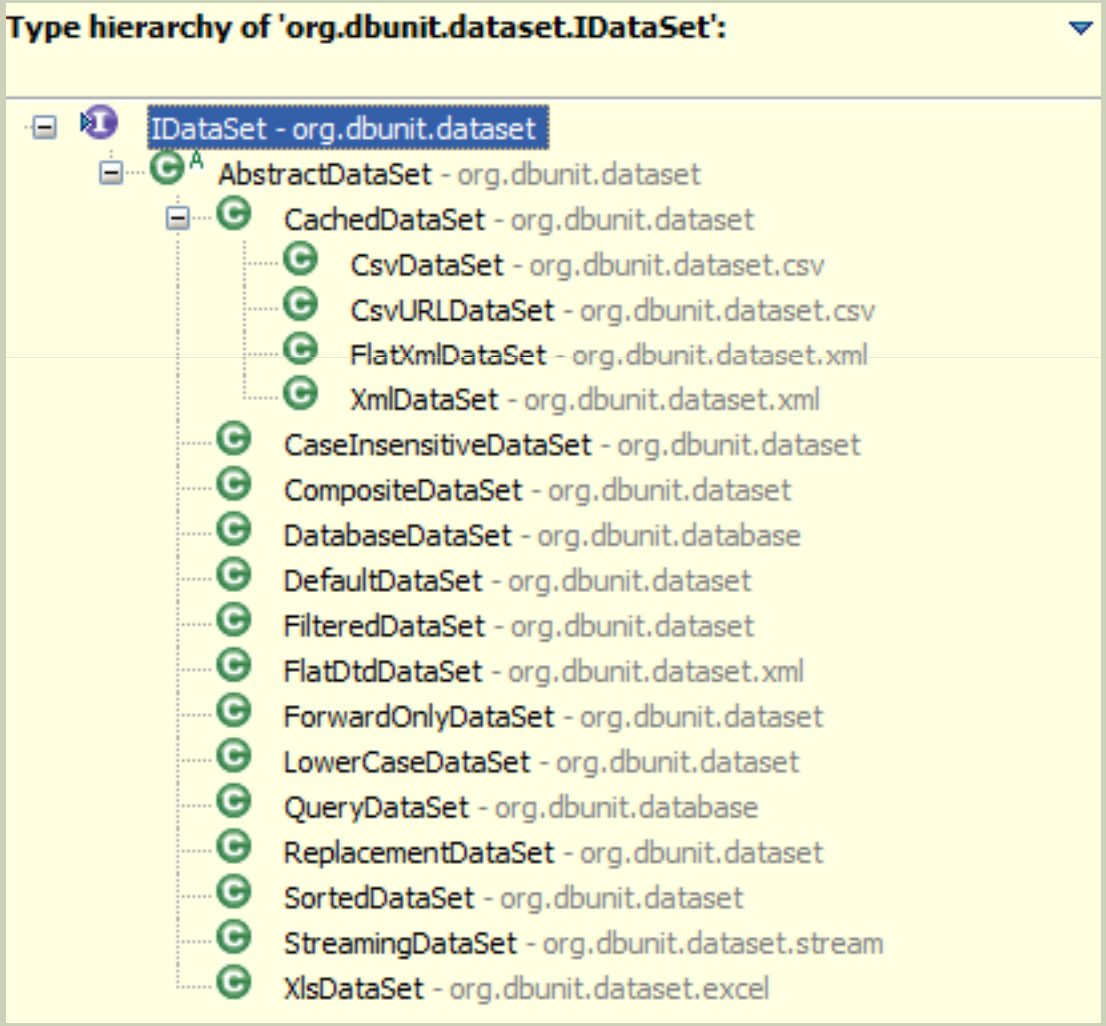
```
@Override
protected IDatabaseConnection getConnection() throws Exception {
    Class.forName("com.mysql.jdbc.Driver");
    Connection connection = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/treinamento","root", "");
    IDatabaseConnection mySqlConnection = new  DatabaseConnection(connection);
    mySqlConnection.getConfig().setProperty(DatabaseConfig.PROPERTY_DATATYPE_FACTORY,
            new MySqlDataTypeFactory());
    return mySqlConnection;
}
```

GETDATASET

- É um método abstrato da classe DatabaseTestCase
- Deve retornar um objeto que implemente a interface IDataSet
- É responsável por retornar o conjunto de dados que deve ser utilizado para povoar o banco de dados.

```
protected IDataSet getDataSet() throws Exception {
    return new FlatXmlDataSetBuilder().build(new FileInputStream("input.xml"));
}
```

IDATASET


Type hierarchy of 'org.dbunit.dataset.IDataSet':
IDataSet - org.dbunit.dataset
AbstractDataSet - org.dbunit.dataset
CachedDataSet - org.dbunit.dataset
CsvDataSet - org.dbunit.dataset.csv
CsvURLDataSet - org.dbunit.dataset.csv
FlatXmlDataSet - org.dbunit.dataset.xml
XmlDataSet - org.dbunit.dataset.xml
CaseInsensitiveDataSet - org.dbunit.dataset
CompositeDataSet - org.dbunit.dataset
DatabaseDataSet - org.dbunit.database
DefaultDataSet - org.dbunit.dataset
FilteredDataSet - org.dbunit.dataset
FlatDtdDataSet - org.dbunit.dataset.xml
ForwardOnlyDataSet - org.dbunit.dataset
LowerCaseDataSet - org.dbunit.dataset
QueryDataSet - org.dbunit.database
ReplacementDataSet - org.dbunit.dataset
SortedDataSet - org.dbunit.dataset
StreamingDataSet - org.dbunit.dataset.stream
XlsDataSet - org.dbunit.dataset.excel

O ARQUIVO XML

- Possui todos os registros que devem estar povoados na base de dados no momento dos testes
- Graças à flexibilidade do XML, pode-se criar tags com o nome das tabelas
- Dentro destas tags, pode-se criar atributos que correspondem aos atributos da tabela na base de dados
- Cada atributo com o valor respectivo que deve ser povoado no banco

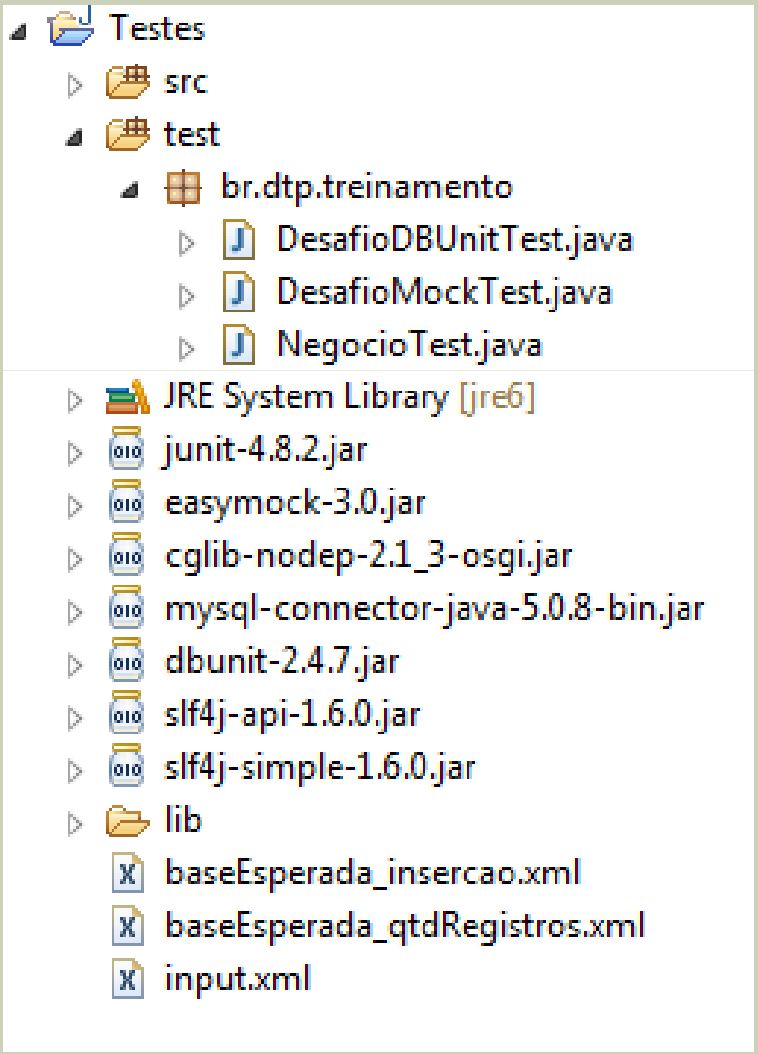
EXEMPLO DO XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
    <!--  Tabela Entidade 1 -->
    <entidade1 id="1" nome="Nome da entidade 1"/>
    <entidade1 id="2" nome="Nome da entidade 2"/>

    <!--  Tabela Entidade 2 -->
    <entidade2 id="1" quantidade="0"/>
    <entidade2 id="2" quantidade="1"/>
    <entidade2 id="3" quantidade="10"/>
    <entidade2 id="4" quantidade="1000"/>

    <!--  Tabela Entidade 3 -->
    <entidade3 />

</dataset>
```

AINDA SOBRE O XML

- Não há limite para a quantidade de tabelas
- Cada tabela pode ter quantos registros (linhas no XML) sejam necessários
- Caso uma tabela não seja utilizada, pode-se inserir uma tag “sem atributos” para esta tabela
  - Isso fará que os dados da tabela sejam apagados
  - Caso a tabela não seja informada no XML, os registros da mesma não serão afetados
- O local padrão do arquivo XML é o diretório raiz do projeto

TUDO NO SEU LUGAR



MÃOS À OBRA

- Adicionar as *libs* no *classpath* do projeto
  - dbunit-2.4.8.jar
  - slf4j-api-1.6.0.jar
  - slf4j-simple-1.6.0.jar
  - hsqldb.jar
- Criar a classe “EntidadeDAODBUnitTest” estendendo a classe DatabaseTestCase
  - Criar um método de testes vazio para a classe poder ser executada
- Criar o arquivo XML
- Fazer testes com o XML, adicionando/removendo registros e limpando toda a tabela

DADOS DINÂMICOS

- O DbUnit também permite inserir valores dinâmicos nos dados em XML

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
    <!--  Tabela Entidade -->
    <entidade id="1" nome="Entidade numero 1" data_inicial="2014-01-01" data_final="2015-01-01"
         email="wcaquino@gmail.com"
        numero_documento="123456789" tipo_documento="1" data_gravacao="ontem"/>

    <entidade id="2" nome="Entidade numero 2"  data_inicial="null" data_final="null"
    email="email@email.com"
        numero_documento="11223000121" tipo_documento="2" data_gravacao="hoje"/>
</dataset>
```

REPLACEMENT DATASET

```
    @Override
    protected IDataSet getDataSet() throws Exception {
//        return new FlatXmlDataSetBuilder().build(new FileInputStream("input.xml"));

        IDataSet dataSet = new FlatXmlDataSetBuilder().build(new FileInputStream("inputReplaced.xml"));

        ReplacementDataSet rDataSet = new ReplacementDataSet(dataSet);
        rDataSet.addReplacementObject("null", null);
        rDataSet.addReplacementObject("hoje", new Date());
        Calendar ontem = Calendar.getInstance();
        ontem.add(Calendar.DAY_OF_MONTH, -1);
        rDataSet.addReplacementObject("ontem", ontem.getTime());

        return rDataSet;
    }
```

ASSERTION

- Métodos da classe Assertion

```
public class Assertion
{
    public static void assertEquals(ITable expected, ITable actual)
    public static void assertEquals(IDataSet expected, IDataSet actual)
}
```

- ITable: Representa a coleção de uma tabela
- IDataSet: Representa uma coleção de tabelas

TESTANDO UMA OPERAÇÃO

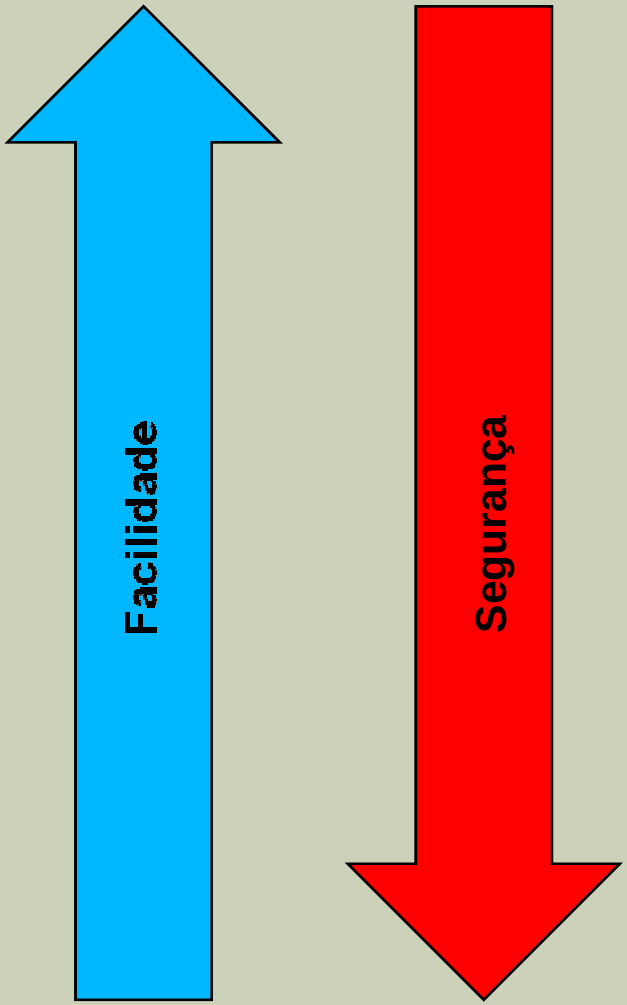
- Teoricamente, após efetuar uma operação, o estado da base de dados deve ter sido alterado.
- Para garantir que a operação foi realizada com sucesso, deve-se checar que o estado atual da base de dados deve refletir as alterações realizadas pela última operação.

- Para realizar esta checagem, deve ser efetuada uma consulta. Exemplos:
  - Ao inserir: consulta o registro inserido
  - Ao alterar: consulta os dados alterados
  - Ao excluir: A consulta ao registro deve estar vazia

- Ou seja:
  - ***EstadoAtual = EstadoAnterior + operaçãoRealizada***

ESTRATÉGIAS PARA TESTAR O BANCO

- Usar dados conhecidos
- Utilização dos métodos já implementados para coletar as informações
- Coletar as informações diretamente no banco (SQL, JPQL, framework de persistência utilizado, etc...)
- Utilização de massa de dados prevista



USANDO DADOS CONHECIDOS

- Baseado no comportamento desejado no teste, informar diretamente o resultado.

```
public void testQuantidadeRegistrosComDadoConhecido() throws Exception{
    int qtdRegistros = persistencia.getQuantidadeRegistros();
    assertEquals(2, qtdRegistros);
}
```

USANDO MÉTODOS IMPLEMENTADOS

```
public void testQuantidadeRegistrosComMetodosImplementados() throws Exception{
    int qtdRegistros = persistencia.getQuantidadeRegistros();

    int qtdRegistrosEsperado = persistencia.retrieveAll().size();

    assertEquals(qtdRegistrosEsperado, qtdRegistros);
}
```

COLETAR INFORMAÇÕES DIRETAMENTE NO BANCO

```java
public void testQuantidadeRegistrosNaMao() throws Exception{
    int qtdRegistros = persistencia.getQuantidadeRegistros();

    //possibilidade 1
    ITable estadoAtualTabela = getConnection().createQueryTable("resultado",
                "SELECT count(*) \"count\" FROM " + TABELA_ENTIDADE);

    int qtdRegistrosEsperado = (Integer) estadoAtualTabela.getValue(0, "count");
    assertEquals(qtdRegistrosEsperado, qtdRegistros);

    //possibilidade 2
    estadoAtualTabela = getConnection().createQueryTable("resultado",
            "SELECT * FROM " + TABELA_ENTIDADE);
    qtdRegistrosEsperado = estadoAtualTabela.getRowCount();
    assertEquals(qtdRegistrosEsperado, qtdRegistros);

    //possibilidade 3
    PreparedStatement stmt = getConnection().getConnection()
        .prepareStatement("SELECT count(*) FROM " + TABELA_ENTIDADE);
    ResultSet rs = stmt.executeQuery();
    rs.next();
    qtdRegistrosEsperado = rs.getInt(1);
    assertEquals(qtdRegistrosEsperado, qtdRegistros);
}
```

Se tiver usando algum framework de persistencia, pode usá-lo também

MASSA DE DADOS PREVISTA

```
public void testQuantidadeRegistrosComMassaDeDadosPrevista() throws Exception{
    int qtdRegistros = persistencia.getQuantidadeRegistros();

    IDataSet dadosEsperados = new FlatXmlDataSet(
            new FlatXmlProducer(new InputSource("baseEsperada_qtdRegistros.xml")));
    ITable tabelaEsperada = dadosEsperados.getTable(TABELA_DESAFIO);
    int qtdRegistrosEsperado = tabelaEsperada.getRowCount();

    assertEquals(qtdRegistrosEsperado, qtdRegistros);
}
```

DADOS VOLÁTEIS

- Geralmente os id's das tabelas são um problema
  - Na maioria dos casos, estes são incrementados automaticamente
  - Para comparações do estado atual da tabela com um estado desejado (informado em XML), todos os campos devem estar iguais
  - Porém o id de um registro presumidamente recém inserido será diferente a cada execução
  - Devemos sempre atualizar o XML?

COLUMN FILTERS

- Uma solução para este problema é a filtragem de atributos.
- Desta forma, apenas os atributos relevantes serão analisados

```
public void testInsert() throws Exception{
    Entidade entidade = UmaEntidade().deNome("Exemplo").comCPFNumero(123).criar();

    BancoUtils.registrosEntidade();
    persistencia.salvar(entidade);
    BancoUtils.registrosEntidade();

    IDataSet expectedDataSet = new FlatXmlDataSetBuilder().build(new FileInputStream("baseEsperada_insercao.xml"));
    ITable expectedTable = expectedDataSet.getTable(TABELA_ENTIDADE);
    ITable tabelaEsperadaFiltrada = DefaultColumnFilter.includedColumnsTable(expectedTable,
            new String[] {"nome", "data_inicial", "data_final", "numero_documento", "tipo_documento", "email"});

    IDataSet databaseDataSet = getConnection().createDataSet();
    ITable actualTable = databaseDataSet.getTable(TABELA_ENTIDADE);
    ITable tabelaAtualFiltrada = DefaultColumnFilter.excludedColumnsTable(actualTable,
            new String[] {"id", "data_gravacao"});

    Assertion.assertEquals(tabelaEsperadaFiltrada, tabelaAtualFiltrada);
}
```

BOAS PRÁTICAS

- Utilizar uma base de dados por desenvolvedor
- Não se preocupe em deixar “rastros”
- Use vários arquivos XML pequenos
- Se os dados são utilizados apenas para consulta, não tem necessidade de “resetar” a base após cada teste, basta uma vez pela Classe/Suíte

EXPORTANDO UM BANCO

```
IDataSet fullDataSet = getConnection().createDataSet();
FileOutputStream xmlStream = new FileOutputStream("bancoFlatExportado.xml");
FlatXmlDataSet.write(fullDataSet, xmlStream);
```

REFACTORING PARA JUNIT 4

```
public class EntidadeDAODBUnitTestRefact {
    private EntidadeDAOInterface persistencia;

    private final static String TABELA_ENTIDADE = "entidade";

    protected IDatabaseConnection getConnection() throws Exception {
        return new DatabaseConnection(ConnectionFactory.getConnection());
    }

    protected IDataSet getDataSet() throws Exception {
        return new FlatXmlDataSetBuilder().build(new FileInputStream("input.xml"));
    }

    @Before
    public void setUp() throws Exception {
        persistencia = new EntidadeDAO();
        DatabaseOperation.CLEAN_INSERT.execute(getConnection(), getDataSet());
    }

    @After
    public void after() throws Exception {
//      DatabaseOperation.DELETE_ALL.execute(getConnection(), getDataSet());
    }

    @Test
    public void testInsert() throws Exception{
        Entidade entidade = UmaEntidade().deNome("Exemplo").comCPFNumero(123).criar();

        persistencia.salvar(entidade);
```

MÃOS À OBRA

- Copiar as classes “EntidadeDAO”, “EntidadeNegocio” para o projeto.
- Criar testes para a classe “EntidadeNegocio”.
  - Salvar deve ser usando o método “Massa de dados prevista”
  - 100% de cobertura na classe testada
- Banco HSQLDb. A conexão está configurada na classe *ConnectionFactory.* Para visualizar os dados do Banco, usar a classe *BancoUtils.*

- Estrutura do Banco
  - TABLE ENTIDADE(
    - ID INTEGER GENERATED BY DEFAULT NOT NULL PRIMARY KEY,
    - NOME VARCHAR(50) NOT NULL,
    - DATA_INICIAL DATE DEFAULT NULL,
    - DATA_FINAL DATE DEFAULT NULL,
    - NUMERO_DOCUMENTO NUMERIC(15) DEFAULT NULL,
    - TIPO_DOCUMENTO NUMERIC(1) DEFAULT NULL,
    - EMAIL VARCHAR(20) DEFAULT NULL,
    - DATA_GRAVACAO DATE NOT NULL)