Documentos

ISSN 1677-9274
Dezembro, 2017

153

# Testes automatizados no sistema BDGF com Arquillian e Junit

Embrapa

*ISSN 1677-9274*
*Dezembro, 2017*

***Empresa Brasileira de Pesquisa Agropecuária***
***Embrapa Informática Agropecuária***
***Ministério da Agricultura, Pecuária e Abastecimento***

# *Documentos 153*

# Testes automatizados no sistema BDGF com Arquillian e Junit

*Fábio Danilo Viera*

Embrapa Informática Agropecuária
Campinas, SP
2017

**Embrapa Informática Agropecuária**
Av. Dr. André Tosello, 209 - Cidade Universitária, Campinas - SP
Fone: (19) 3211-5700
https://www.embrapa.br/informatica-agropecuaria


Supervisão editorial: Kleber X. Sampaio de Souza
Revisão de texto: Adriana Farah Gonzales
Normalização bibliográfica: Maria Goretti G. Praxedes
Editoração eletrônica: Tuíra Santana Favarin, sob supervisão de Flávia Bussaglia Fiorini.
Imagem da capa: Tuíra Santana Favarin

**1ª edição on-line - 2017**



**Dados Internacionais de Catalogação na Publicação (CIP)**
**Embrapa Informática Agropecuária**

Vieira, Fábio Danilo.
Testes automatizados no sistema BDGF com Arquillian e Junit / Fábio Danilo Vieira. - Campinas : Embrapa Informática Agropecuária, 2017.
41 p. : il. - (Documentos / Embrapa Informática Agropecuária , ISSN 1677-9274 ; 153).

1. Java EE. 2. Teste de integração. 3. Teste de interface. 4. Servidor de aplicação. 5. Selenium. 6. Java Server Faces I. Título. II. Embrapa Informática Agropecuária. III. Série.

CDD 21. ed. 005.74

# Autor

**Fábio Danilo Vieira**
Tecnólogo em Processamento de dados, mestre em Engenharia Agrícola, analista da Embrapa Informática Agropecuária, Campinas, SP.

# Apresentação

Realizar o controle da qualidade de um software é um grande desafio devido às diversas dificuldades encontradas durante o processo de desenvolvimento, as quais podem estar relacionadas a questões técnicas, humanas, burocráticas e de negócio. O que se espera de um sistema de software é que ele faça corretamente o que o usuário necessita, além de fazê-lo de forma eficiente, segura e escalável. Tais premissas são muitas vezes confirmadas por meio de testes manuais do sistema após o término de módulos específicos ou até mesmo do sistema inteiro. Esta abordagem manual (e muitas vezes feita de forma isolada) leva à ocorrência de muitos problemas e deve ser evitada. Por isso, a utilização de testes automatizados se torna uma prática altamente recomendável em qualquer processo de desenvolvimento de software.

No desenvolvimento de programas em Java, há diversos frameworks responsáveis por testes automatizados na linguagem. O mais famoso deles é o JUnit, que foi idealizado para execução de testes de códigos Java. Devido à sua grande versatilidade, muitos desenvolvedores utilizam-no em conjunto com outros frameworks para criação de testes funcionais automatizados mais robustos. Um desses frameworks é o Arquillian, que pode ser utilizado em conjunto com JUnit para construir testes em sistemas de software com recursos do Java Enterprise Edition (Java EE).

O Arquillian é uma plataforma de testes funcionais e de integração, que pode ser usada para testar a camada de negócio em Java. Seu principal objetivo é tornar os testes de integração e funcionais tão simples quanto escrever testes unitários. Arquillian é a ferramenta ideal para testar aplicações com recursos do Java EE, pois cuida para que a aplicação seja implantada em um contêiner (servidor web ou de aplicação) durante o teste, e que também sejam injetados os objetos a serem testados diretamente na classe de teste.

Dessa forma, durante o desenvolvimento do sistema Banco de Dados de Genótipos e Fenótipos (BDGF), que é um sistema desenvolvido com recursos do Java EE, optou-se por utilizar o Arquillian em conjunto com o JUnit para criação do ambiente de testes automatizados. Neste documento, será mostrado como configurar os arquivos necessários para execução desses frameworks e também como criar um arquivo de testes numa classe de negócio nesse ambiente

*Sílvia Maria Fonseca Silveira Massruhá*
Chefe Geral da Embrapa Informática Agropecuária

# Sumário

# Testes automatizados no sistema BDGF com Arquillian e Junit

*Fábio Danilo Vieira*

## Introdução

De forma geral, ao se desenvolver a funcionalidade de um software, realiza-se a análise do problema, procura-se por uma possível solução e, por fim, codifica-se essa solução. Após essas etapas, numa abordagem tradicional de desenvolvimento de software, normalmente o desenvolvedor implementa testes manuais para verificar se está tudo funcionando como deveria. Nesses testes, erros são comumente detectados, sendo necessário que os desenvolvedores os corrijam e refaçam o conjunto de testes novamente. Ademais, utiliza-se ainda mais uma etapa de verificação de erros, que é a submissão do software a um processo de avaliação de qualidade. Esse processo geralmente ocorre com o uso de testes manuais executados por usuários e pelos próprios desenvolvedores (TASSEY, 2002).

Na implementação e execução desses testes, diversos problemas são ocasionados devido aos testes serem manuais, tais como criação de sistemas com vasta quantidade de erros, atraso na sua entrega, dificuldade de manutenção, etc. Repetir a execução de uma extensa lista de testes de forma manual é uma atividade muito cansativa, sendo frequente que os testadores não identifiquem novos erros a cada mudança importante no código. Esses erros não identificados (ou

identificados de forma tardia) acabam gerando prejuízos para todos os atores do projeto, tanto para o cliente, que é prejudicado pelos atrasos nos prazos pré-estabelecidos e com a desconfiança na qualidade do software, quanto para os desenvolvedores, que perdem um tempo precioso para corrigir os erros. O pior é que este ciclo tende a se repetir até que a manutenção do software se torne algo tão complicado que valerá a pena reconstruí-lo por completo (BERNARDO; KON, 2008).

Felizmente, não é preciso se recorrer apenas aos testes manuais para verificar o correto funcionamento de um sistema. Para auxiliar a equipe de desenvolvedores nesta tarefa, existem os testes automatizados, que são programas simples que verificam as funcionalidades de um software por meio de testes pré-estabelecidos em código. Uma das principais vantagens desta abordagem é que os testes podem ser repetidos com grande facilidade e a qualquer momento. A possibilidade de se reproduzir testes que simulem ocorrências específicas garantem que requisitos importantes do sistema não sejam esquecidos ou ignorados por erro humano (BERNARDO; KON, 2008).

Além do mais, pelo fato de ser um programa que será executado pelo computador, é possível codificar testes mais sofisticados e abrangentes do que testes realizados de forma manual. É possível, por exemplo, simular a inserção de milhares de registros num banco de dados ou até mesmo o acesso simultâneo de milhares de usuários no sistema, testes impossíveis de serem feitos de modo manual.

No desenvolvimento do sistema BDGF a equipe responsável optou, inicialmente, por realizar apenas testes manuais nele. Porém, com o passar do tempo, novas funcionalidades foram adicionadas e outras antigas completamente modificadas, ficando inviável para a equipe tomar conta do desenvolvimento de novos códigos, de corrigir erros detectados e testar novamente todo o software. Diante disso, foi decidido que se utilizariam testes de unidade e de integração automatizados no controle de erros e qualidade do sistema BDGF.

O BDGF é um sistema web desenvolvido na linguagem de programação

Java sob a arquitetura Java EE, a qual consiste de um conjunto de serviços e de interfaces de programa de aplicação, *Application Programming Interface* (API) para o desenvolvimento de sistemas corporativos. Em softwares mais simples, como os que não utilizam arquitetura Java EE, por exemplo, normalmente os testes podem ser feitos utilizando bibliotecas amplamente conhecidas, como JUnit, que é um *framework* construído para criação e manutenção de testes unitários de códigos Java, possuindo características da própria linguagem. Entretanto, para sistemas mais complexos, ou seja, que utilizam arquitetura Java EE, por exemplo, deve-se utilizar *frameworks* adicionais para a realização dos testes automatizados, pois estes irão cuidar para que a aplicação seja executada corretamente no servidor de aplicação e que todos os objetos sejam injetados de modo transparente. Dessa forma, durante o desenvolvimento do sistema BDGF, optou-se por utilizar o Arquillian em conjunto com o JUnit para criação do ambiente de testes automatizados.

## Tipos de testes

Para diversos tipos de erros existem seus testes específicos, os quais realizam verificações seguindo determinados padrões e premissas. Assim, é de extrema importância classificar os casos de teste por tipo, devido a diversas razões, dentre as quais pode-se citar (BERNARDO, 2011):

- Auxilia na manutenção dos testes e na definição de diferentes cenários para correção de erros.

- Utilização de ferramentas próprias e padronizações diferentes.

- Pelo fato de o tempo de execução variar de um tipo para outro, testes de execução lenta não afetarão o *feedback* rápido da execução de testes mais velozes.

- Favorece a extração das métricas por tipo de teste, que pode ser útil para observar pontos de verificação que precisam de mais esfoço.

Para cada tipo de teste existe um conjunto de ferramentas especializadas, justamente para facilitar a escrita e tornar o código mais enxuto e legível. O que pode ocorrer é utilizar uma ferramenta de um tipo de teste específico para objetivos que não são de sua responsabilidade natural, o que pode levar à criação de testes de baixa qualidade. Dessa forma, é necessário conhecer a diversidade de opções de ferramentas para se optar pelas mais apropriadas para o que se está querendo verificar (BERNARDO, 2011). A seguir, será feita uma breve explanação sobre alguns desses testes (ANICHE et al. 2014; BERNARDO, 2011):

**Teste de unidade** irá testar uma única unidade do sistema. Este tipo fará o teste de maneira isolada, normalmente simulando as prováveis dependências que aquela unidade tem. Em sistemas orientados a objetos, é comum que a unidade seja uma classe. Ou seja, quando se deseja escrever testes de unidade para a classe Pessoa, por exemplo, a bateria de testes testará o funcionamento da classe Pessoa de forma isolada, sem interações com outras classes.

**Teste de integração** é aquele que irá testar a integração entre duas partes de um sistema. Um teste que se codifica para a classe FuncionariosDao, por exemplo, onde esse teste acessará o banco de dados, é um teste de integração, pois está se testando a integração do software com diversos componentes desse software e também com o SGBD. Os testes que verificam se as classes se comunicam corretamente com serviços web, enviam mensagens via *socket*, entre outros, são considerados testes de integração.

**Teste de interface** verifica a correção de uma interface por meio da simulação de eventos de usuários, como se um usuário estivesse controlando dispositivos como mouse e teclado. Dessa forma, a partir dos efeitos advindos dos eventos, são realizadas verificações na interface e em outras camadas para se certificar que a interface está funcionando apropriadamente.

**Teste de sistema** visa garantir que o sistema funciona como um todo,

ou seja, com todas as unidades trabalhando juntas e integralmente. É comumente chamado de teste de caixa preta, pois o sistema é testado utilizando todos seus componentes: banco de dados, serviços web, *batch jobs*, etc.

**Testes de aceitação** são testes onde as equipes verificam se uma determinada funcionalidade está "aceita" ou não. Também conhecido como teste funcional ou de história de usuário, são testes de correção e validação. Esses testes são comumente especificados por clientes ou usuários finais do sistema para verificar se um módulo funciona como foi especificado. Por isso o termo "aceitação", pois ele verifica se o cliente aceita as funcionalidades que foram implementadas. Os testes de aceitação devem utilizar uma linguagem próxima da natural, para evitar problemas de interpretação e de ambiguidades. Nas diretrizes do desenvolvimento ágil, uma história do cliente não será finalizada enquanto os testes de aceitação não certificarem que o sistema atende aos requisitos especificados.

Independentemente do tipo do teste, todos têm suas vantagens e desvantagens. Um teste de unidade, por exemplo, é bastante fácil de ser codificado e pode ser executado de modo muito simples. Entretanto, não é um teste que simula bem o mundo real. Por outro lado, um teste de sistema faz uma simulação bastante real, porém, é muito mais complicado de ser escrito, oferece mais trabalho de manutenção e exige mais tempo para ser executado.

## Sobre o Arquillian

Arquillian é uma plataforma de testes para Java que permite aos desenvolvedores criarem facilmente testes de integração e funcionais automatizados, constituindo-se como uma ferramenta adequada para testar aplicações com recursos do Java EE, pois cuida para que a aplicação seja implantada em um contêiner (servidor web ou de aplicação) durante o teste, e que também sejam injetados os objetos a serem testados diretamente na classe de teste. O Arquillian alcança todos os pontos possíveis que um teste unitário simples não consegue

alcançar, fornecendo uma plataforma para integração dos componentes dentro do ambiente de execução (ALLEN et al., 2017).

Basicamente, o Arquillian incorpora uma parte do JUnit (ou TextNG)[1] para executar casos de teste em um contêiner Java (por exemplo: GlassFish[2] e WildFly[3]). Além disso, o Arquillian trabalha sozinho com todo o controle de contêiners, implantação dos projetos e inicialização do servidor, liberando o desenvolvedor para que se concentre apenas na lógica de negócio em que o teste está sendo codificado.

O Arquillian trabalha em conjunto com os *frameworks* de teste mais conhecidos do Java (por exemplo, JUnit), permitindo que os testes sejam criados utilizando uma *Integrated Development Environment* (IDE) de desenvolvimento (como NetBeans[4]), por exemplo, sem a necessidade de incluir qualquer outro módulo de software. Utiliza também, em sua arquitetura, a biblioteca ShrinkWrap para definir quais arquivos serão utilizados, realizando o empacotamento das classes e seus recursos dependentes. Em seguida, o Arquillian implanta e executa todos esses arquivos empacotados (geralmente num arquivo em formato jar - Java ARchive) no contêiner destino. Após a execução, os resultados dos testes são capturados, e no final, o Arquillian desimplanta o arquivo, exibindo os resultados na IDE para o desenvolvedor por meio do *framework* de testes (por exemplo, JUnit). A Figura 1 exibe a composição da arquitetura do Arquillian.

[1] *Disponível em: <http://testng.org>.*
[2] *Disponível em: <https://javaee.github.io/glassfish>.*
[3] *Disponível em: <https://wildfly.org>.*
[4] *Disponível em: <https://netbeans.org>.*


**Figura 1.** Arquitetura da plataforma Arquillian.
Fonte: Allen et al. (2017).

De forma resumida, o Arquillian procura minimizar a carga sobre o desenvolvedor na realização dos testes de integração, incluindo (ALLEN et al., 2017):

- Gerenciamento do ciclo de vida do contêiner (start/stop).
- Agrupando a classe de teste com classes e recursos dependentes em um arquivo implementável (utilizando o ShrinkWrap).
- Enriquecendo a classe de teste, solucionando injeções de dependência e outros serviços do Java EE (por exemplo, @Inject, @EJB e @Resource, etc.).
- Implementando o arquivo no contêiner para testes (deploy/undeploy).
- Capturando resultados e falhas e os devolvendo para o framework de testes (JUnit, por exemplo).

## Material e métodos de desenvolvimento do sistema BDGF

O sistema BDGF foi desenvolvido com o intuito de oferecer uma solução que fosse eficiente tanto no armazenamento quanto na consulta de dados genotípicos, fenotípicos e de pedigree de animais. Para tanto, utilizou-se um diagrama de dados inicialmente proposto por (HIGA; OLIVEIRA, 2015), que propunha um modelo de uso geral em sistemas relacionados a armazenamento de dados dessa natureza na Embrapa. Esse diagrama foi redesenhado de forma que possibilitasse a implementação do tipo *JavaScript Object Notation* (JSON) em campos de tabelas relacionadas a fenótipos e do tipo texto (*character varying*) em campos de tabelas relacionadas a genótipos do diagrama. Com a implementação do tipo JSON e do tipo texto nessas tabelas, foi possível o uso da abordagem *Not Only SQL* (NoSQL)[5] para armazenar parte dos dados sem que seja necessário se importar com a sua normalização, agilizando consultas que necessitariam realizar junções (*joins*) com outras tabelas.

# Material

O sistema BDGF está sendo codificado em computadores dedicados ao desenvolvimento e está hospedado em servidores virtuais para testes e homologação. Para o desenvolvimento do sistema, escolheu-se componentes de tecnologia de informação (TI) disponíveis no mercado, ou seja, dentro da filosofia do uso de software livre. Primeiramente, como sistema operacional, optou-se pelo Linux Ubuntu[6] em todos os equipamentos.

Como SGBD usou-se o PostgreSQL[7], versão 9.5. O PostgreSQL foi escolhido por ser um SGBD confiável, amplamente utilizado no

[5] *Disponível em: <http://nosql-database.org>.*
[6] *Disponível em: <https://www.ubuntu.com>.*
[7] *Disponível em: <https://www.postgresql.org>.*

mercado e pelo qual a equipe de desenvolvimento tem conhecimentos mais avançados. Outrossim, possui em suas últimas versões, a partir da 9.2, o formato JSON para tipificar os campos de tabelas. A tecnologia JSON consiste num formato de padrão aberto que consiste de conjuntos de pares na forma "chave:valor". Ela foi utilizada para que fosse possível viabilizar o uso da abordagem NoSQL, para armazenar parte dos dados sem que seja necessário se importar com a sua normalização.

O software utilizado para controle de versão foi o Subversion (SVN), versão 1.8.8. A linguagem de programação escolhida foi Java[8], versão 8, e seus componentes da tecnologia Java Enterprise Edition (Java EE), que consiste de um conjunto de serviços e de interfaces de API para o desenvolvimento de sistemas corporativos.

Dentre as tecnologias Java EE disponíveis e utilizadas pelo BDGF destaca-se, entre outras, a estrutura Java Server Faces (JSF). A arquitetura do framework JSF emprega o modelo *Model, View, Controller* (MVC), que faz a separação entre as camadas de apresentação e de aplicação. Na sua implementação como modelo MVC, o JSF possui uma camada de visualização bem distinta do conjunto de classes de modelo (Figura 2). O JSF ainda se destaca por ser uma especificação do Java EE, isto é, todo servidor de aplicações Java tem que vir com uma implementação do *framework*.

[8] *Disponível em: <https://www.oracle.com/br/java/>.*

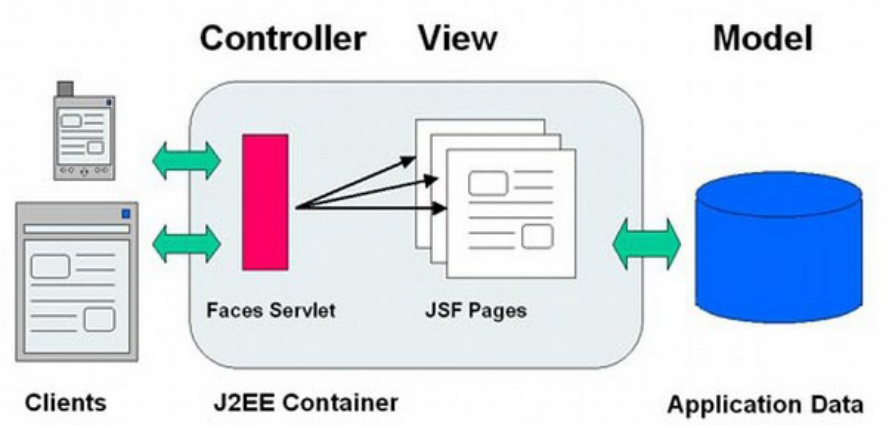


**Figura 2.** Ilustração do modelo MVC (Model View Controller) adotado pelo JSF
Fonte: Soares (2014).

Todo sistema Web desenvolvido em Java que segue a especificação Java EE deve ser instalado e executado dentro de um servidor de aplicação. O servidor escolhido para abrigar o sistema BDGF foi o WildFly[9], versão 10, que é um dos servidores de aplicação mais seguros do mundo.

A aplicação foi desenvolvida na IDE NetBeans (versão 8.2), que pode ser executada em diversas plataformas, como Windows, Linux, Solaris e MacOS. Essa versão do NetBeans já oferece o pacote JDK, versão 8, como linguagem Java padrão, além de variados recursos para se criar aplicativos profissionais para Web.

[9] *Disponível em: <http://wildfly.org/downloads/>.*

# Métodos

Os casos de uso e diagramas de sequência foram desenhados a partir da lista de requisitos obtida para desenvolvimento do sistema de banco de dados, elaborada por meio da identificação do problema e das necessidades dos usuários. A partir disso, o projeto de desenvolvimento do sistema seguiu os conceitos do Scrum, que é um *framework* ágil para a realização de projetos complexos. O Scrum destaca-se dos demais métodos ágeis pela maior ênfase dada ao gerenciamento do projeto. Reúne atividades de monitoramento e *feedback*, em geral, por meio de reuniões rápidas e diárias com toda a equipe, procurando identificar e corrigir quaisquer deficiências no processo de desenvolvimento (SCHWABER, 2004).

O método Scrum serve-se de fundamentos como: equipes pequenas (no máximo, sete pessoas); requisitos desconhecidos e iterações curtas, dividindo o desenvolvimento em intervalos de tempos pequenos (no máximo, trinta dias), também chamados de Sprints. Ademais, existem três personagens importantes nesse processo: *Product Owner* (descreve os interesses de todos no projeto), Time (desenvolve as funcionalidades do produto) e *ScrumMaster* (responsável por garantir que todos sigam as práticas do Scrum).

O início de um projeto Scrum ocorre com uma visão geral do produto, a qual contém todas as características e restrições do produto determinadas pelo cliente (SCHWABER, 2004). Em seguida, cria-se o *Product Backlog* contendo a lista de todos os requisitos conhecidos, sendo então priorizados e divididos em tarefas, ou *sprints* (Figura 3).

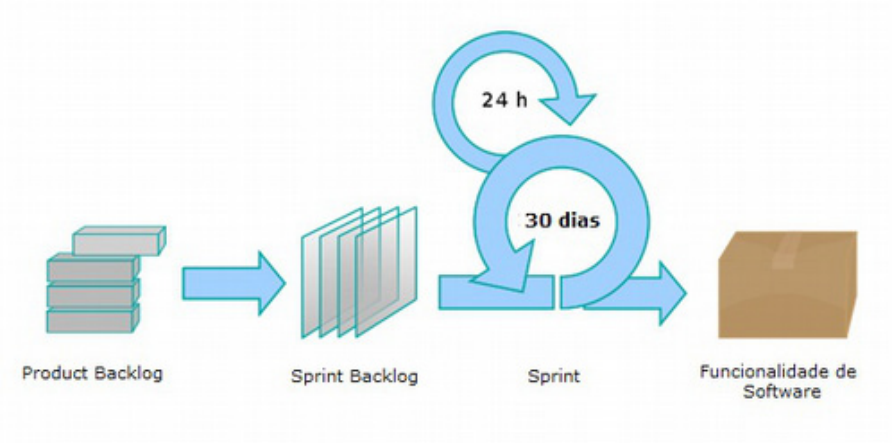


**Figura 3.** Representação geral do funcionamento Scrum.
Fonte: Desenvolvimento... (2017?).

Nesse contexto, a fase de construção compreendeu várias iterações (*sprints*), nas quais, em cada iteração, procurava-se desenvolver e corrigir pequenas partes do sistema, sendo esses testados e integrados no final, procurando satisfazer um subconjunto de requisitos do projeto. Tendo completado todas as tarefas do *backlog*, o desenvolvimento encontra-se agora em fase de homologação que acontece por meio de interações diretas com os usuários, nas quais o código fonte e a estrutura do sistema estão sendo ajustados e estabilizados.

Como ponto inicial do desenvolvimento do sistema, partiu-se pelo diagrama de dados inicialmente proposto por (HIGA; OLIVEIRA, 2015). Depois de diversas análises e levando-se em consideração que o SGBD utilizado seria o PosgreSQL e que a quantidade de dados a ser armazenada seria sempre crescente, optou-se por utilizar o formato JSON (*jsonb*) em campos de tabelas relacionadas a fenótipos e indivíduos (animais), e formato texto (*character varying*) em campos da tabela de genótipos do diagrama.

## Criando um teste automatizado no sistema BDGF utilizando Arquillian

Nesta seção será mostrado como criar testes de integração no sistema BDGF utilizando EJB, CDI e JPA com o apoio do *framework* Arquillian por meio do gerenciador de projetos Maven, bem como executá-los no WildFly. O Apache Maven é uma ferramenta de gerenciamento de projetos de software baseado no conceito de um modelo de objeto de projeto (POM) para gerenciar a construção de um projeto, realizando a automação da compilação de projetos Java (ou desenvolvidos em outras linguagens) (MAVEN, 2017).

Projetos baseados no Maven utilizam um arquivo XML (pom.xml) para descrever como o projeto de software será construído, as dependências e *plug-ins* necessários, forma de empacotamento (formato jar, war, etc.), entre outros. O Maven realiza o *download* de bibliotecas Java e seus plug-ins dinamicamente de um ou mais repositórios e armazena-os em uma área de *cache* local (MAVEN, 2017).

Dessa forma, como o sistema BDGF foi desenvolvido por meio da ferramenta Maven, o arquivo pom.xml teve que ser corretamente configurado para baixar as dependências e *plug-ins* necessários para uso do Arquillian no projeto. Abrindo o arquivo pom.xml, foi primeiramente acrescentado o código da Figura 4 para que o projeto Maven realizasse os testes com o Arquillian.

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.jboss.arquillian</groupId>
            <artifactId>arquillian-bom</artifactId>
            <version>1.1.13.Final</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

**Figura 4.** Trecho para importação do BOM (Bill of Materials) do Arquillian.

Esse fragmento de XML foi inserido diretamente acima do elemento <build> para importar o BOM (*Bill of Materials*) do Arquillian, ou uma matriz de versões para as dependências transitivas do *framework*.

Em seguida, o seguinte trecho da Figura 5 foi inserido dentro da seção <dependencies>:

```
<!-- JUnit -->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
<!-- Arquillian JUnit Container -->
<dependency>
    <groupId>org.jboss.arquillian.junit</groupId>
    <artifactId>arquillian-junit-container</artifactId>
    <scope>test</scope>
</dependency>
```

**Figura 5.** Dependências para JUnit e API de integração do Arquillian com JUnit.

Esse trecho, na primeira dependência, faz o download do JUnit, versão 4.12, e a segunda dependência faz a integração do Arquillian com JUnit, além de adicionar as bibliotecas do Arquillian e do ShrinkWrap para o classpath de testes. Essas bibliotecas são necessárias para escrever e compilar um teste Arquillian integrado com JUnit.

O próximo trecho (Figura 6) contêm o link para o contêiner que será baixado pelo Arquillian:

```
<dependency>
    <groupId>org.wildfly</groupId>
    <artifactId>wildfly-arquillian-container-managed</artifactId>
    <version>8.2.0.Final</version>
    <scope>test</scope>
</dependency>
```

**Figura 6.** Dependência do contêiner Wildfly gerenciável. A versão é do pacote, e não do servidor.

Como o sistema BDGF utiliza o WildFly como servidor de aplicação (contêiner) para *deploy* e execução, o Arquillian deve utilizar o mesmo servidor e mesma versão para a criação dos testes automatizados, de preferência. No caso do BDGF, a versão do WildFly utilizada é a 10.

Sendo assim, no pom.xml foi adicionado esse trecho de dependência para realizar o download do contêiner do WildFly gerenciado. A versão 8.2 refere-se à versão da dependência, e não do servidor em si.

Como configuração final no pom.xml, deve-se inserir (ou editar, caso já exista no arquivo XML) o seguinte trecho da Figura 7 dentro da seção <plugins>:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-dependency-plugin</artifactId>
    <executions>
        <execution>
            <id>unpack</id>
            <phase>process-test-classes</phase>
            <goals>
                <goal>unpack</goal>
            </goals>
            <configuration>
                <artifactItems>
                    <artifactItem>
                        <groupId>org.wildfly</groupId>
                        <artifactId>wildfly-dist</artifactId>
                        <version>10.1.0.Final</version>
                        <type>zip</type>
                        <overWrite>false</overWrite>
                        <outputDirectory>${project.build.directory}</outputDirectory>
                    </artifactItem>
                </artifactItems>
            </configuration>
        </execution>
    </executions>
</plugin>
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.17</version>
    <configuration>
        <!-- Fork every test because it will launch a separate AS instance -->
        <systemPropertyVariables>
            <java.util.logging.manager>org.jboss.logmanager.LogManager</java.util.logging.manager>
            <jboss.home>/opt/teste_wildfly-10.1.0.Final</jboss.home>
            <module.path>/opt/teste_wildfly-10.1.0.Final/modules</module.path>
        </systemPropertyVariables>
        <redirectTestOutputToFile>false</redirectTestOutputToFile>
    </configuration>
</plugin>
```

**Figura 7.** Plug-ins e suas configurações para uso do Arquillian no projeto.

Esses *plug-ins* são necessários para fase de compilação e testes do sistema BDGF. Geralmente, o primeiro *plug-in* (responsável pela descompactação dos artefatos do projeto) está presente por *default* nos projetos criados no NetBeans. Dessa maneira, foi preciso realizar apenas uma pequena modificação para se adequar ao Arquillian. Já o segundo *plug-in* refere-se a ponte entre o Arquillian e o contêiner onde serão executados os testes.

Para finalizar a configuração, foi necessário criar um arquivo chamado “arquillian.xml” no diretório \src\test\resources com o conteúdo da Figura 8:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
To change this license header, choose License Headers in Project Properties.
To change this template file, choose Tools | Templates
and open the template in the editor.
-->
<arquillian xmlns="http://jboss.org/schema/arquillian"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://jboss.org/schema/arquillian
       http://jboss.org/schema/arquillian/arquillian_1_0.xsd">

    <container qualifier="wildfly10" default="true">
          <configuration>
              <property name="jbossHome">/opt/teste_wildfly-10.1.0.Final</property>
              <property name="managementPort">10990</property>
              <property name="javaVmArguments">
                  -Djboss.socket.binding.port-offset=1000
              </property>
              <property name="allowConnectingToRunningServer">true</property>
          </configuration>
    </container>
</arquillian>
```

**Figura 8.** Arquivo de configuração arquillian.xml do sistema BDGF.

Neste caso, o arquivo está apontando o caminho do servidor WildFly (target/wildfly-10.1.0.Final), versão 10.1, onde será implantado e executado qualquer teste automatizado utilizando o Arquillian no sistema BDGF. Além disso, o arquivo configura argumentos para a máquina virtual do Java que irá rodar o servidor, como a memória utilizada e as portas nas quais o servidor para testes poderá executar, o que é importante na medida em que o servidor de produção normalmente é executado na porta 8080.

Agora, o sistema está pronto para se criar um teste automatizado. Qualquer teste deve ser criado dentro do “Pacotes de Teste” (ou no caminho do projeto, ou seja, /bdgf/src/test) que a IDE NetBeans exibe na janela de navegação do projeto aberto atualmente.

Para se criar um teste automatizado, pode-se clicar com o botão direito do mouse em qualquer uma das classes existentes no projeto e depois em “Ferramentas > Criar/Atualizar Testes”, como na Figura 9:

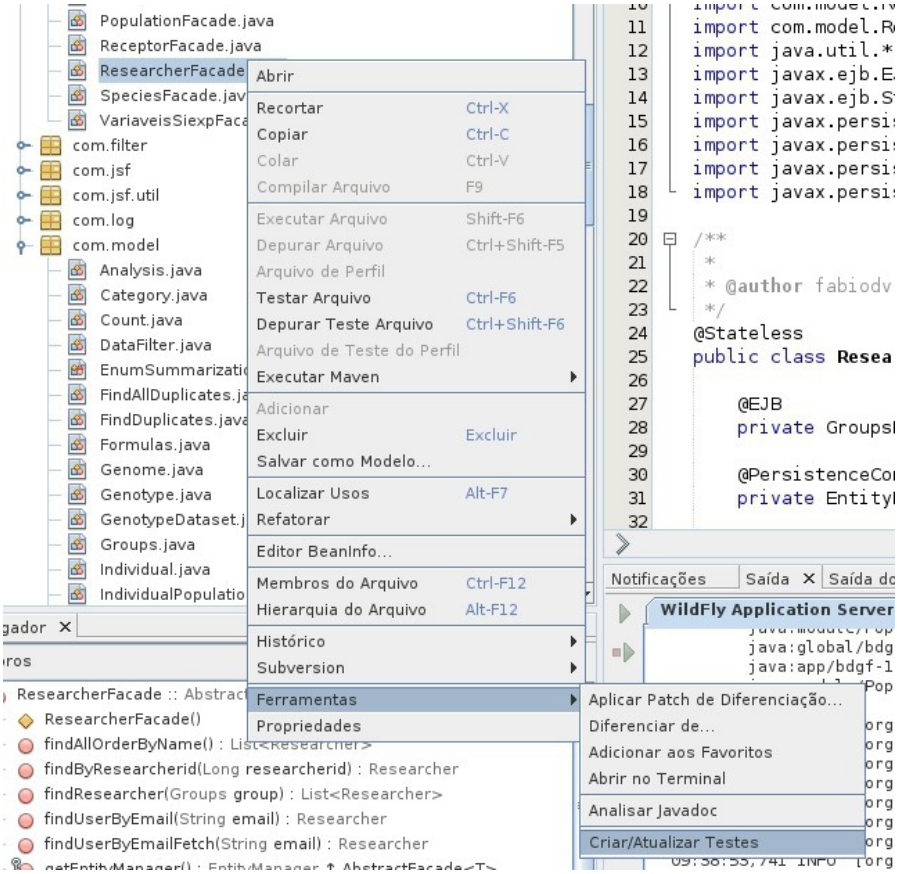


**Figura 9.** Criando um teste automatizado para uma classe no NetBeans.

Nesse caso, foi escolhida a classe de negócio ResearcherFacade (classe relativa a usuários e pesquisadores do sistema BDGF. Após o clique no submenu para "Criar/Atualizar Testes", a seguinte janela (Figura 10) para criação do teste será exibida:

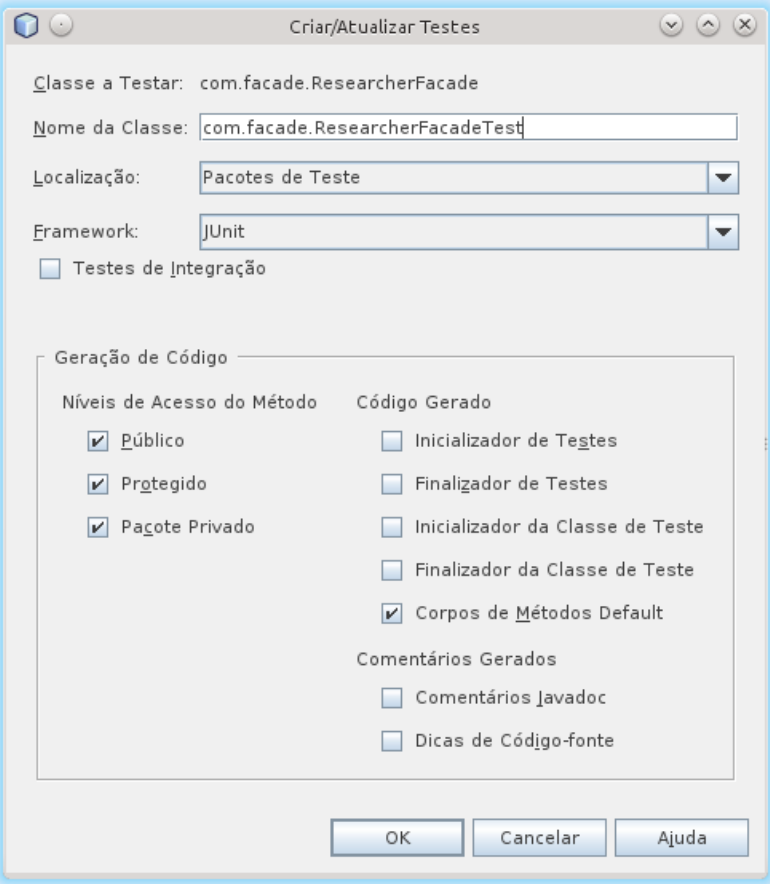


**Figura 10.** Janela para definição de nome da classe de teste e sua estrutura de conteúdo.

Pode-se preservar o nome de classe sugerido pelo NetBeans, assim como a Localização e o *framework* padrão, que é o JUnit mesmo. O restante pode ser deixado como mostrado na Figura 10.

Após clicar no botão Ok, será gerado a classe de teste com o formato padrão do JUnit, de acordo com o que foi definido na janela anterior. No caso da classe ResearcherFacade, foi gerado um código com

testes para todos os seus métodos. Como padrão, todos iniciam-se com o prefixo “test”. A seguir, na Figura 11, parte desse código é demonstrada:

```
package com.facade;

import com.model.Groups;
import com.model.Researcher;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import javax.ejb.embeddable.EJBContainer;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 *
 * @author m316380
 */
public class ResearcherFacadeTest {

    public ResearcherFacadeTest() {
    }

    @Test
    public void testRegisterLog() throws Exception {
        System.out.println("registerLog");
        Researcher entity = null;
        Enum Operations = null;
        Researcher researcher = null;
        Groups group = null;
        EJBContainer container = javax.ejb.embeddable.EJBContainer.createEJBContainer();
        ResearcherFacade instance =
                (ResearcherFacade)container.getContext().lookup("java:global/classes/ResearcherFacade");
        instance.registerLog(entity, Operations, researcher, group);
        container.close();
        fail("The test case is a prototype.");
    }

    @Test
    public void testGetActiveUser() throws Exception {
        System.out.println("getActiveUser");
        EJBContainer container = javax.ejb.embeddable.EJBContainer.createEJBContainer();
        ResearcherFacade instance =
                (ResearcherFacade)container.getContext().lookup("java:global/classes/ResearcherFacade");
        Researcher expResult = null;
        Researcher result = instance.getActiveUser();
        assertEquals(expResult, result);
        container.close();
        fail("The test case is a prototype.");
    }
```

**Figura 11.** Parte do código da classe de teste ResearcherFacadeTest.

Para utilizar o Arquillian, grande parte deste código foi descartada. A primeira modificação foi inserir a anotação “@RunWith(Arquillian.class)” acima do nome da classe (Figura 12):

```
@RunWith(Arquillian.class)
public class ResearcherFacadeTest {
```

**Figura 12.** Anotação para definir que o Arquillian será o gerenciador desse teste.

Em seguida, a anotação @Deployment e o método createDeployment() foram adicionados logo abaixo do nome da classe (Figura 13). Essa anotação e esse método definem como o Arquillian irá realizar a implantação do código (formato do arquivo de empacotamento, inclusive) e quais classes deverão ser empacotadas (por meio da subrotina addPackages), ou seja, esses métodos informam ao ShrinkWrap para incluir esses arquivos dentro da aplicação "<nome_app>.war". Essas dependências são para a execução do projeto inteiro, pois, ao inserir apenas a classe que seria testada, o Arquillian "reclamava" de dependências faltantes que tinham alguma relação com a classe ou com algum pacote que foi adicionado anteriormente.

```
@Deployment
public static Archive createDeployment() {
    return ShrinkWrap.create(WebArchive.class)
        .addPackages(true, "org.hibernate")
        .addPackages(true, "org.primefaces")
        .addPackages(true, "com.github")
        .addPackages(true, "com.vladsch")
        .addPackages(true, "com.unboundid")
        .addPackages(true, "org.apache")
        .addPackages(true, "br.embrapa")
        .addPackages(true, "com.facade")
        .addPackages(true, "com.model")
        .addPackages(true, "com.jsf")
        .addPackages(true, "com.log")
        .addPackages(true, "com.filter")
        .addPackages(true, "com.converter")
        .addClass(ResearcherFacade.class)
        .addAsResource("META-INF/persistence.xml")
        .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
}
```

**Figura 13.** Anotação @Deployment e método createDeployment para a classe de teste ResearcherFacadeTest.

Agora é possível usufruir do poder que o Arquillian proporciona. É possível, por exemplo, escrever testes utilizando os recursos do Java EE 7 normalmente, dentro de uma classe de teste. No caso do exemplo que está se demonstrando, foi adicionada a anotação @EJB acima da variável researcherFacade para se utilizar a injeção de dependência do

EJB (a injeção por CDI também pode ser utilizada por meio da anotação @Inject), como mostra o código a seguir (Figura 14):

```
@EJB
ResearcherFacade researcherFacade;
```

**Figura 14.** Anotação @EJB para injeção de dependência de variável.

Finalmente, foram escritas duas funções de testes para dois métodos da classe ResearcherFacade (Figura 15). Existiam outros métodos, mas apenas esses dois foram considerados mais cruciais para serem testados. Lembrando que não é necessário (e nem se deve) criar procedimentos de testes para todo e qualquer método de uma classe.

```
@Test
public void testIsValidLogin() throws Exception {
    System.out.println("isValidLogin");
    String userName = "m123456";
    String password = "abcde12345";
    Researcher result = researcherFacade.isValidLogin(userName, password);
    assertNotNull(result);
}


@Test
public void testFindUserByEmail() throws Exception {
    System.out.println("findUserByEmail");
    String email = "jose.silva@embrapa.br";
    Researcher result = researcherFacade.findUserByEmail(email);
    assertNotNull(result);
}
```

**Figura 15.** Funções de teste criadas para dois métodos da classe ResearcherFacade.

O primeiro teste verifica se o método *isValidLogin* está fucionando corretamente com o usuário e senha passados como parâmetros. Já o segundo teste irá verificar se o e-mail colocado como parâmetro para o método *findUserByEmail* está cadastrado no banco. A função *assertNotNull* espera um retorno não nulo, pois os valores passados como parâmetros foram pré-validados como corretos e existentes no banco de dados.

Cabe ressaltar aqui que o banco de dados utilizado é específico para testes, mas não apenas para os testes automatizados, mas também para os testes manuais que ainda são realizados pela equipe. Para o

sistema de homologação, existe uma máquina virtual específica com o sistema e um banco de dados em separado. Quanto ao ambiente de desenvolvimento, até o presente momento, os testes estão codificados no mesmo projeto de desenvolvimento do sistema BDGF, inclusive os de interface, que utilizam apenas o *framework* Selenium. Há a intenção de, num futuro próximo, colocá-los num ambiente próprio para testes, por questões de organização e facilidade de manutenção.

Dessa forma, como ainda estão num mesmo ambiente, após a compilação e construção do sistema inteiro, se tudo correr bem com os testes (ver abaixo de Results que não houve erros nos testes), a seguinte saída (Figura 16) será gerada e o arquivo de implantação bdgf.war será criado para execução.

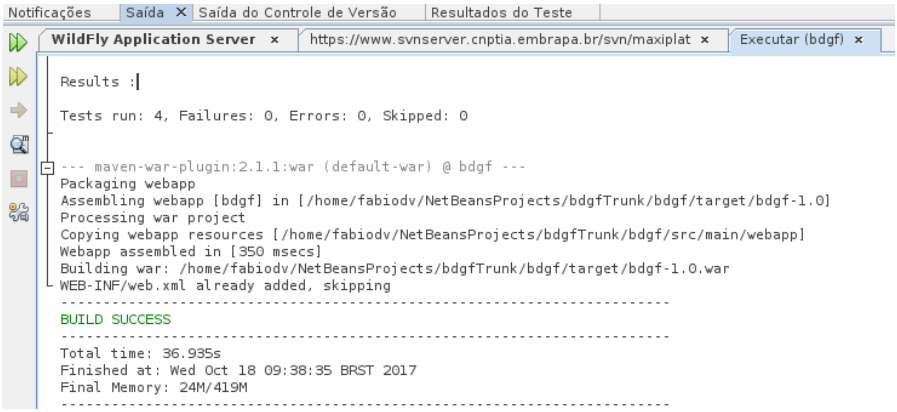


**Figura 16.** Saída gerada após compilação e construção com sucesso nos testes automatizados.

Se ocorrer (em) alguma (s) falha (s) na compilação e construção do sistema, a seguinte saída (Figura 17) será exibida, alertando que houve erro(s) nos testes. Com isso, o arquivo bdgf.war não é gerado e o *deploy*, por consequência, não ocorre.

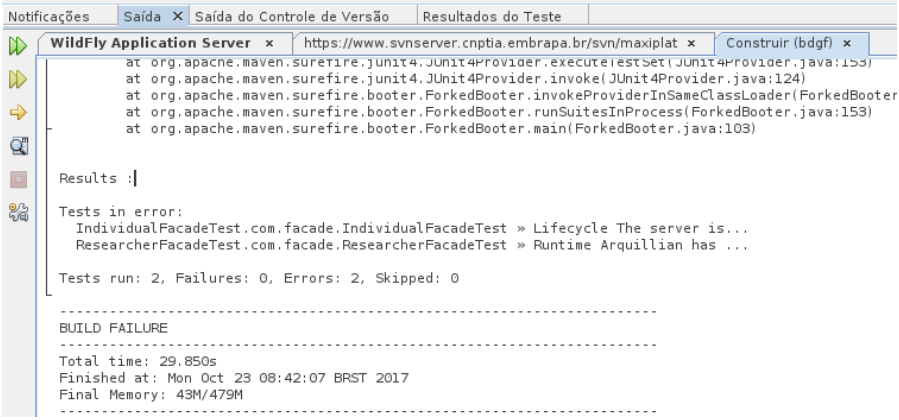


**Figura 17.** Saída gerada após compilação e construção com falha nos testes automatizados.

É possível realizar apenas o teste do projeto, sem ter de empacotar e fazer o deploy dos arquivos compilados como demonstrado anteriormente. Para tanto, utlizando o ambiente NetBeans, basta clicar no menu Executar > Testar Projeto (Figura 18).


**Figura 18.** Menu para apenas testar o projeto.

Caso os testes ocorram sem erros, a seguinte tela com cabeçalho na cor verde é exibida (Figura 19).

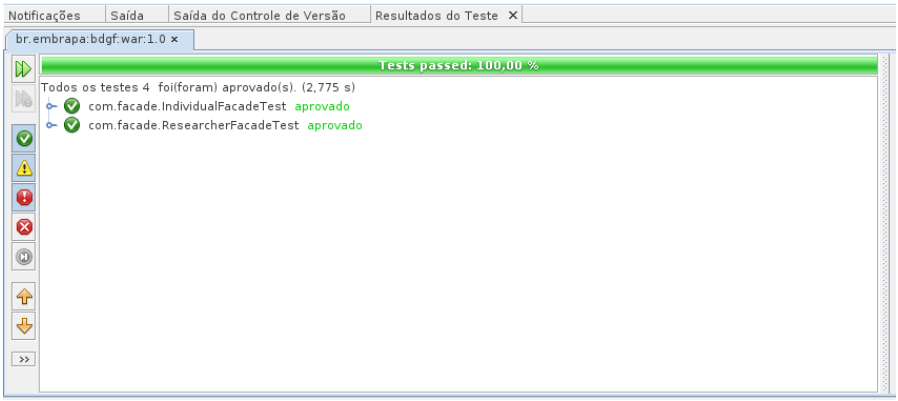


**Figura 19.** Tela demonstrando que os testes ocorreram de forma correta.

Por outro lado, caso os testes ocorram com erros, a seguinte tela com cabeçalho na cor vermelha é mostrada (Figura 20).

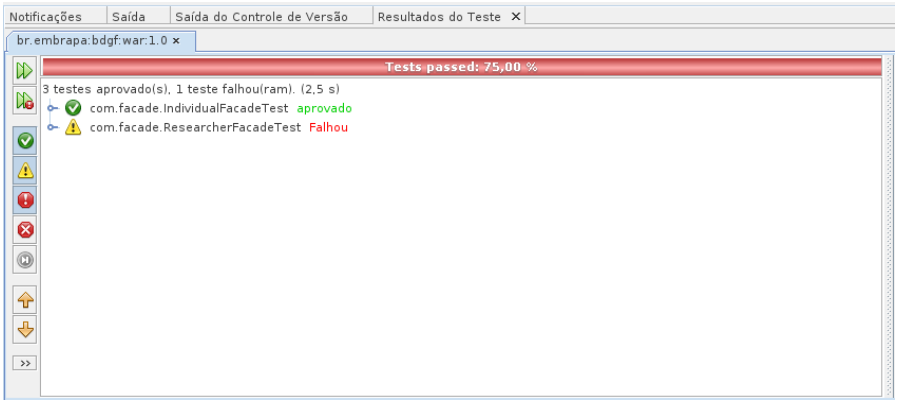


**Figura 20.** Tela demonstrando que ocorreram erros nos testes.

Como visto, diversos testes automatizados podem ser criados utilizando o *framework* Arquillian. Por ser possível construir testes que abrangem desde o acesso ao banco de dados até verificação de injeção de dependências, testes completos podem ser realizados. Ou se os parâmetros de um método foram alterados com um simples teste de asserção. Enfim, há uma infinidade de possibilidades, e a "criatividade" do desenvolvedor pode auxiliar muito na elaboração de

testes poderosos que permitam a construção e integração contínua de um projeto Java EE.

## Resultados e Discussão

Até o presente momento, os testes desenvolvidos para o sistema BDGF, utilizando Arquillian, são do tipo integração. Assim, por enquanto, a parte do código que está sendo coberta pelos testes envolve apenas classes que acessam o banco de dados do software. Pela experiência vivenciada pela equipe no uso da ferramenta Arquillian, os testes de integração mostraram ser mais apropriados de serem implementados para as necessidades atuais do sistema, que são verificações de correção nos acessos ao banco de dados. Tentou-se utilizar o *framework* Arquillian para codificação de testes de interface, porém, dificuldades encontradas na configuração de dependências (conflitos de versões, principalmente) do projeto e no desenvolvimento do próprio código inviabilizaram a continuação na codificação desse tipo de teste, optando-se por utilizar apenas a biblioteca Selenium[10] para escrita de testes de interface. A seguir, um exemplo de teste de interface utilizando apenas Selenium (Figura 21):

[10] *Disponível em: <http://www.seleniumhq.org>.*

```
/**
 *
 * @author m316380
 */
public class TesteLoginIT {

    @Test
    public void testSimple() throws Exception {
        // Create a new instance of the Firefox driver
        // Notice that the remainder of the code relies on the interface,
        // not the implementation.

        //System.setProperty("webdriver.chrome.driver", "/home/xxx/chromedriver");
        System.setProperty("webdriver.gecko.driver", "/home/xxx/geckodriver");

        // *** não mostrar navegador na tela ***
        FirefoxBinary firefoxBinary = new FirefoxBinary();
        firefoxBinary.addCommandLineOptions("-headless");
        FirefoxOptions options = new FirefoxOptions()
        .setProfile(new FirefoxProfile());
        options.setBinary(firefoxBinary);
        // ***

        WebDriver driver = new FirefoxDriver(options);

        // And now use this to visit NetBeans
        driver.get("http://licv059.cnptia.embrapa.br/bdgf/");
        // Alternatively the same thing can be done like this
        // driver.navigate().to("http://www.netbeans.org");

        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

        WebElement email = driver.findElement(By.name("formLogin:email"));
        WebElement password = driver.findElement(By.name("formLogin:password"));
        WebElement signIn = driver.findElement(By.name("formLogin:btnLogin"));

        email.sendKeys("xxxx");
        password.sendKeys("yyyy");
        signIn.sendKeys(Keys.ENTER);

        //Close the browser
        driver.quit();
    }

}
```

**Figura 21.** Exemplo de código de teste de interface utilizado apenas Selenium (sem Arquillian).

Outro item observado foi o tempo adicional gasto com os testes para compilação e construção do sistema, que atualmente é algo muito irrelevante (questão de segundos), o que não está comprometendo em nada a disponibilização e a avaliação do sistema por parte dos usuários. Obviamente que este tempo tende a aumentar com a criação de novos testes, ainda mais com os de interface, mas os benefícios que podem ser colhidos devem suplantar qualquer tempo a mais na construção e implantação do software, até mesmo porque a equipe

conta com ferramentas como o Jenkins[11] para auxiliar nesse processo de integração contínua do sistema (Figura 22). Uma alternativa que está sendo estudada pela equipe é implementar os testes de interface em outro projeto, pois assim eles não interfeririam na compilação e construção do sistema BDGF, seja em termos de tempo ou de erros que poderão ocorrer nesse tipo de teste.

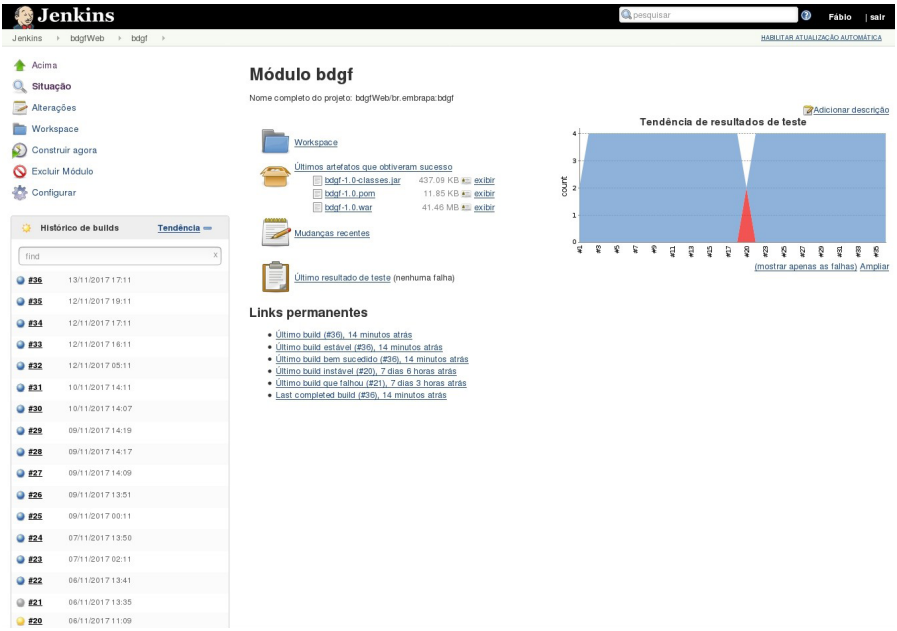


**Figura 22.** Ferramenta Jenkins utilizada para auxiliar na integração contínua do sistema.

Vale ressaltar que, até um certo ponto do sistema, boa parte do código foi escrita sem a codificação de testes automatizados, utilizando-se apenas do expediente de testes manuais para garantia de correção de código, o que gerou diversos transtornos. O uso do *framework* Arquillian incentivou a equipe a ter (e retomar) o hábito de desenvolver códigos de testes logo após a implementação de qualquer nova funcionalidade no sistema, seja essa nova implementação uma classe ou mesmo um método mais complexo. Com isso, também se reforçou

[11] *Disponível em: <https://jenkins.io>.*

no desenvolvedor a necessidade da adoção de boas práticas na escrita de um código, como tratamento de erros, criação de métodos e variáveis com nomes mais significativos, melhor formatação do código, etc., enfim, passos que muitas vezes são deixados de lado quando se desenvolve sem estar atrelado à necessidade de se escrever um teste automatizado para o código em questão. Além disso, menos inconsistências no acesso ao banco de dados do sistema foram encontradas, o que ocorria frequentemente com o uso de testes manuais.

Prentende-se, enfim, implementar testes que contemplem a maior parte possível do código do sistema BDGF, sejam testes que utilizem ou não o *framework* Arquillian. Contudo, no contexto do desenvolvimento desse sistema, o Arquillian abriu um caminho que auxiliará e muito no árduo trajeto de desenvolvimento de um sistema web mais robusto e confiável.

## Considerações Finais

Quando se utiliza apenas testes manuais no desenvolvimento de um sistema, diversos problemas podem surgir, como a criação de um software com muitos erros não identificados, atraso na entrega desse sistema, dificuldade de manutenção, etc. Esses problemas acabam gerando prejuízos para o cliente e também para os desenvolvedores, que perdem muito tempo corrigindo-os e causando desconfiança nos usuários sobre a segurança do produto. No caso do sistema BDGF, isso era muito frequente, pois se aplicava apenas esse expediente de testes. Pensando em evitar esse tipo de situação por muito tempo, foram desenvolvidos testes automatizados para verificação, principalmente, de erros de software e inconsistências no banco de dados. Como o sistema BDGF utiliza-se de recursos do Java EE, foi necessário usar o *framework* Arquillian, juntamente com o já conhecido pacote JUnit, para construção das classes de testes. Com o Arquillian ficou mais fácil desenvolver testes na arquitetura Java EE, pois é uma ferramenta que cuida sozinha para que a aplicação seja implantada em um contêiner e que também sejam injetados os objetos a serem testados diretamente

na classe de teste. Diversos testes podem ser desenvolvidos, desde verificar acesso ao banco de dados até cuidar para que injeção de dependências esteja funcionando corretamente. Além disso, testes podem ser realizados na estrutura do banco de dados por meio de simples SQL's para constatar se o mesmo foi modificado. Há muitas possibilidades para criação de testes com o Arquillian, e seu gerenciamento automático de recursos Java EE permite que o desenvolvedor cuide apenas do código em si, sem se preocupar com o restante do ambiente.

# Referências

ALLEN, D.; KNUTSEN, A.; MUIR, P.; RUBINGER A. **Arquillian**: an integration testing framework for Java EE. Disponível em: <https://docs.jboss.org/arquillian/reference/1.0.0.Alpha1/en-US/html_single/>. Acesso em: 19 out. 2017.

ANICHE, M. **Blog da Caelum**: unidade, integração ou sistema? qual teste fazer? 2014. Disponível em: <http://blog.caelum.com.br/unidade-integracao-ou-sistema-qual-teste-fazer/>. Acesso em: 13 nov. 2017.

BERNARDO, P. C. **Padrões de testes automatizados**. 2011. 197 p. Dissertação (Mestrado) – Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo.

BERNARDO, P. C.; KON, F. A importância dos testes automatizados - controle ágil, rápido e confiável de qualidade. **Engenharia de Software Magazine**, Rio de Janeiro, v. 1, n. 3, p. 54-57, jul. 2008.

DESENVOLVIMENTO ágil com Scrum: uma visão geral. <https://www.devmedia.com.br/desenvolvimento-agil-com-scrum-uma-visao-geral/26343>. Acesso em: 25 out. 2017.

HIGA, R. H.; OLIVEIRA, G. B. de. **Banco de Dados de Genótipos e**

**Fenótipos (BDGF) para suporte a estudos de associação genômica ampla e seleção genômica em programas de melhoramento animal**. Campinas: Embrapa Informática Agropecuária, 2015. 30 p. (Embrapa Informática Agropecuária. Documentos, 133). Disponível em: < http://ainfo.cnptia.embrapa.br/digital/bitstream/item/138127/1/Doc133.pdf>. Acesso em: 13 nov. 2017.

MAVEN. **Apache Maven Project**, 2017. Disponível em: <https://maven.apache.org>. Acesso em: 20 de out. 2017.

SCHWABER, K. **Agile project management with Scrum**. Redmond: Microsoft Press. 2004. 163 p.

SOARES, L. **Backing beans em JSF**. [2014]. Disponível em: <http://luissoares.com/jsf-parte-3-backing-beans/>. Acesso em: 13 nov. 2017.

TASSEY, G. **"The economic impacts of inadequate infrastructure for software testing"**: final report. Gaithersburg: National Institute of Standards and Technolog, 2002. 309 p. (RTI Project number 7007.011).

Embrapa

Informática Agropecuária

MINISTÉRIO DA
AGRICULTURA, PECUÁRIA
E ABASTECIMENTO

BRASIL
GOVERNO FEDERAL

CGPE 14132