DOCUMENTOS

171

ISSN 1677-9274
Novembro/2020

Utilizando Selenium para construir testes automatizados no sistema Banco de Dados de Pedigree, Fenótipos e Genótipos

Embrapa
Informática Agropecuária

ISSN 1677-9274
Novembro/2020

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

DOCUMENTOS 171

Utilizando Selenium para construir testes automatizados no sistema Banco de Dados de Pedigree, Fenótipos e Genótipos

Fábio Danilo Vieira

Autor

Embrapa Informática Agropecuária
Campinas, SP
2020

Exemplares desta publicação podem ser adquiridos na:

Embrapa Informática Agropecuária

Av. Dr. André Tosello, 209 - Cidade Universitária
Campinas, SP, Brasil
CEP. 13083-886
Fone: (19) 3211-5700
www.embrapa.br
www.embrapa.br/fale-conosco/sac

Comitê Local de Publicações
da Unidade Responsável

Presidente
Stanley Robson de Medeiros Oliveira

Secretária-Executiva
Maria Fernanda Moura

Membros
Adriana Farah Gonzalez, membro nato, Alexandre de Castro, membro indicado, Carla Cristiane Osawa, membro nato, Debora Pignatari Drucker, membro eleito, Ivan Mazoni, membro eleito, João Camargo Neto, membro indicado, Joao Francisco Goncalves Antunes, membro eleito, Magda Cruciol, membro nato.

Revisão de texto
Adriana Farah Gonzalez

Normalização bibliográfica
Carla Cristiane Osawa

Projeto gráfico da coleção
Carlos Eduardo Felice Barbeiro

Editoração eletrônica
Mariana Pilatti sob supervisão de Magda Cruciol

Arte da capa
Magda Cruciol

1ª edição
Versão digital (2020)

Todos os direitos reservados.
A reprodução não autorizada desta publicação, no todo ou em parte,
constitui violação dos direitos autorais (Lei nº 9.610).

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

Vieira, Fábio Danilo,
Utilizando Selenium para construir testes automatizados no sistema Banco de Dados de Pedigree, Fenótipos e Genótipos / Fábio Danilo Vieira. - Campinas : Embrapa Informática Agropecuária, 2020.

PDF (35 p.) : il. color. - (Documentos / Embrapa Informática Agropecuária, ISSN 1677-9274 ; 171).

1. Automzação de testes. 2. Sistema web. 3. Sistema BDPFG. 4. Selenium. I. Título. II. Embrapa Informática Agropecuária. III. Série.

CDD (21. ed.) 005.74

Carla Cristiane Osawa (CRB-8/10421) © Embrapa, 2020

Autores

Fábio Danilo Vieira

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

Apresentação

A utilização de apenas testes manuais no desenvolvimento de um sistema Web pode ocasionar diversos problemas, tais como a criação de um software com uma quantidade considerável de erros não identificados, atraso na entrega desse produto e dificuldades de manutenção por parte da equipe atual e outros desenvolvedores que vierem a integrarem a mesma. Tais problemas geram prejuízos para o cliente e também para os desenvolvedores, que desperdiçam uma grande quantidade de tempo para corrigi-los, causando desconfiança nos usuários sobre a qualidade e segurança do produto. Dessa forma, o uso de *frameworks* para realizar testes automatizados se torna uma ótima solução para essa situação.

Os *frameworks* de testes automatizados utilizam algoritmos para comparar os resultados reais do programa com os resultados esperados pelo desenvolvedor. O uso dessas bibliotecas facilita o progresso do desenvolvimento de um software, pois funcionam de modo mais rápido e eficiente que o manual. Além disso, podem executar de forma repetitiva, e sem erros, uma enorme quantidade de testes.

O Selenium é provavelmente a solução de código aberto mais amplamente usada para teste de aplicações Web. Selenium é um *framework* que oferece a possibilidade de se criar uma estrutura de testes automatizados de acordo com a funcionalidade de um sistema Web. Além disso, pode ser utilizado em diversas linguagens, como Java (utilizada nesse trabalho), Python, PHP, etc.

Diante desse contexto, durante o desenvolvimento do sistema Banco de Dados de Pedigree, Fenótipos e Genótipos (BDPFG), que é um sistema Web desenvolvido com uso da arquitetura Java EE, decidiu-se utilizar o Selenium (em conjunto com a biblioteca TestNG) para criação de um projeto, em linguagem Java, de testes automatizados. Esse projeto foi estruturado e organizado de forma que outros sistemas Web possam se basear no mesmo para criar um projeto de testes automatizados para suas aplicações Web também. Neste documento, será mostrado como foi estruturado e configurado os pacotes e classes para execução desse projeto de testes para o sistema BDPFG.

Sílvia Maria Fonseca Silveira Massruhá

Chefe-geral

Sumário

Introdução

No desenvolvimento convencional de um software, quando se finaliza o algoritmo de uma funcionalidade requisitada, ou se faz a correção de alguma já existente, normalmente o desenvolvedor realiza testes manuais para verificar se está tudo funcionando como deveria. Durante esses testes, erros são frequentemente detectados. Os desenvolvedores, então, os corrigem e refazem o conjunto de testes manuais novamente. Na execução desses testes, muitos inconvenientes ocorrem devido aos testes serem manuais, tais como atraso na entrega deles, dificuldade de manutenção, etc. Os erros não identificados, ou aqueles identificados tardiamente, geram prejuízos tanto para o cliente, que é prejudicado pela desconfiança na qualidade do software e pelos atrasos nos prazos preestabelecidos, quanto para os desenvolvedores, que desperdiçam um tempo considerável para corrigir os erros.

Felizmente, existem os testes automatizados para auxiliar a equipe de desenvolvedores nesta tarefa. Esses testes são programas de código relativamente simples que verificam as funcionalidades de um software por meio de testes preestabelecidos em código. Uma das características desta abordagem é que os testes podem ser repetidos a qualquer momento. A reprodução de testes automatizados que simula ocorrências específicas garante que requisitos importantes do sistema não sejam esquecidos ou ignorados por erro humano (Bernardo; Kon, 2008).

A implementação de testes automatizados permite criar algoritmos de testes mais complexos e abrangentes do que se fossem realizados de forma manual. Por exemplo, é possível criar um teste que realize a inserção de milhares de registros num banco de dados ou até mesmo o login simultâneo de milhares de usuários no sistema, testes esses impossíveis de serem feitos de modo manual.

No desenvolvimento do sistema BDPFG, a equipe responsável decidiu, inicialmente, por realizar apenas testes manuais. Porém, com o passar do tempo, novas funcionalidades foram adicionadas e outras antigas completamente modificadas, ficando inviável manter o desenvolvimento de novos códigos, de corrigir erros e testar todo o software a cada novo ciclo de implementações e correções. Diante disso, foi decidido que se utilizaria testes automatizados no controle de erros e qualidade do sistema BDPFG.

Pelo fato do BDPFG ser um sistema web desenvolvido na linguagem de programação Java sob a arquitetura Java EE, que consiste em um conjunto de serviços e de interfaces de programa de aplicação, Application Programming Interface (API) para o desenvolvimento de sistemas corporativos, em princípio optou-se por utilizar o *framework* Arquillian em conjunto com a biblioteca Junit[1] para criação do ambiente de testes automatizados. Entretanto, após alguns meses tentando implementar os testes por meio do Arquillian (Vieira, 2017), a equipe percebeu que a tarefa seria muito árdua e muito pouco eficiente, desistindo, assim, da ferramenta. Procurou-se, então, por outras bibliotecas em Java que pudessem facilitar a criação de testes automatizados para o sistema Web BDPFG. Lembrando que, em sistemas web, esses testes costumam ser ainda mais trabalhosos, pois dependem de diversos fatores, como o navegador utilizado, a conexão de rede, sistema operacional, etc. Durante a pesquisa, descobriu-se que existem ferramentas que podem ajudar muito nestes casos, como a biblioteca Selenium.[2]

O Selenium é uma biblioteca livre e gratuita que permite realizar testes automatizados em uma aplicação web de forma automatizada. O Selenium está presente em diversas linguagens, tais como:

[1] Disponível em: <https://junit.org/junit5/>.

[2] Disponível em: <http://selenium.dev/projects/>.

Java, Python, Ruby, PHP, Perl, etc., sendo, possivelmente, a solução mais utilizada em testes automatizados para plataforma web (The Selenium…, 2020). Sua codificação em Java (foco desse trabalho) é relativamente simples e fácil de ser implementada, não sendo necessária a adição de uma diversidade de bibliotecas extras nem a configuração de arquivos XML específicos.

Nesse documento, será mostrado, de forma sucinta, como foram implementados os testes Selenium no sistema BDPFG.

Descrição das ferramentas usadas para construir o sistema BDPFG

O objetivo do sistema BDPFG (Figuras 1 e 2) é ser uma solução eficiente tanto no armazenamento quanto na consulta de dados genotípicos, fenotípicos e de pedigree de animais. Em seu desenvolvimento, utilizou-se um diagrama de dados proposto inicialmente por Higa e Oliveira (2015), que sugeria um modelo de dados de uso geral em sistemas relacionados a armazenamento de dados dessa natureza na Empresa Brasileira de Pesquisa Agropecuária (Embrapa). Esse diagrama foi redesenhado diversas vezes 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

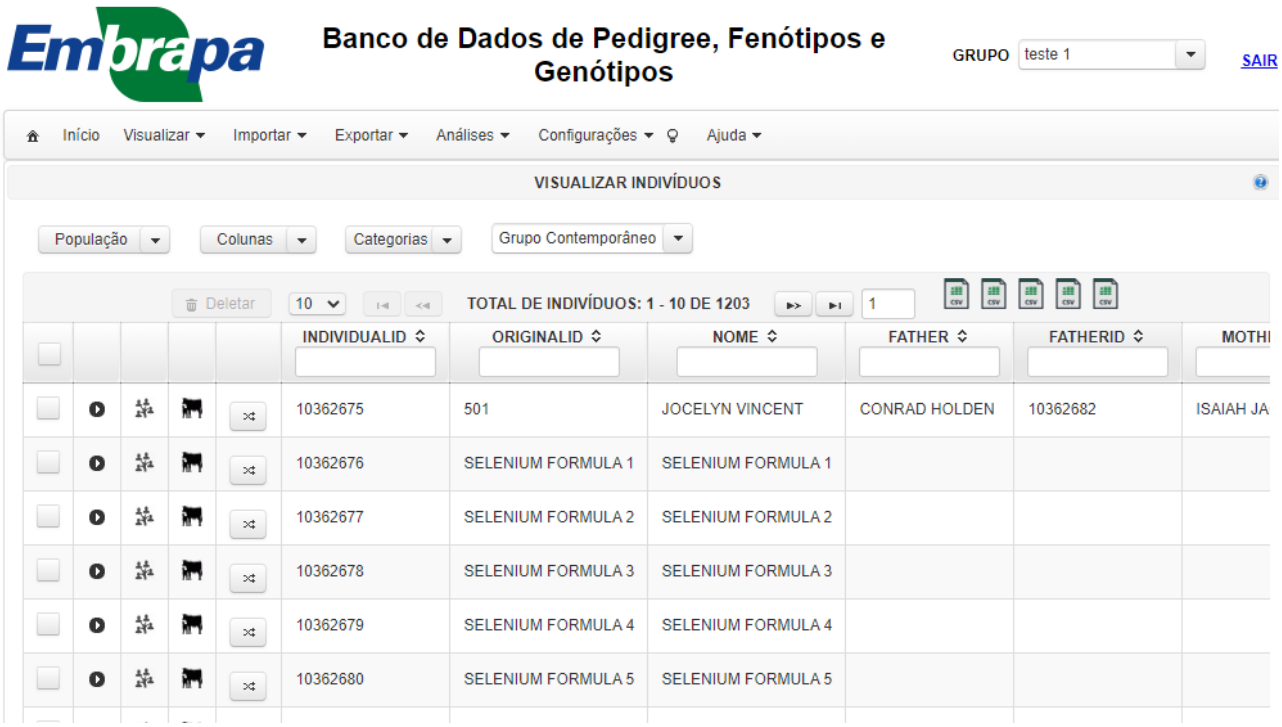


Figura 1. Funcionalidade visualizar indivíduos do sistema BDPFG.

Figura 2. Funcionalidade importar indivíduos do sistema BDPFG

e do tipo texto nessas tabelas, foi possível o uso da abordagem *Not Only SQL* (NoSQL)[3] para armazenar parte dos dados sem que seja necessário se importar com a normalização destes, agilizando consultas que necessitariam realizar junções (joins) com outras tabelas..

O sistema gerenciador de banco de dados (SGBD) utilizado foi o PostgreSQL[4]. O PostgreSQL foi escolhido por ser um SGBD confiável, amplamente utilizado no mer-cado e pelo qual a equipe de desenvolvimento tem conhecimentos mais avançados. Desde as suas últimas versões, a partir da 9.2, implementa 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 normalização dos mesmos.

O software escolhido para controle de versão foi o GitLab. A linguagem de programação utiliza-da foi Java[5], versão 8, e seus componentes da tecnologia Java Enterprise Edition (Java EE), que consiste de um conjunto de serviços e de interfaces de programa de aplicação (API) para o desenvolvimento de sistemas corporativos.

Dentre as tecnologias Java EE disponíveis e utilizadas pelo BDPFG 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 (Soares, 2014). Na sua implementação como modelo MVC, o JSF possui uma camada de visualização bem distinta do conjunto de classes de modelo. 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*. O servidor de aplicação instalado para executar o sistema BDPFG foi o WildFly[6], versão 10, que é um dos servidores de aplicação mais seguros do mundo.

[3] Disponível em: <http://nosql-database.org>.

[4] Disponível em: <https://www.postgresql.org>.

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

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

O sistema BDPFG foi integralmente desenvolvido utilizando-se a IDE NetBeans[7], que pode ser executada em diversas plataformas, como Linux, Windows e MacOS. A versão utilizada do NetBeans no desenvolvimento do sistema já oferece o pacote JDK, versão 8, como linguagem Java padrão, além de variados recursos para se criar aplicativos profissionais para Web.

Metodologia de desenvolvimento do sistema BDPFG

O projeto de desenvolvimento do sistema procurou seguir alguns dos conceitos do Scrum, que é um *framework* ágil para a realização de projetos complexos. Ele destaca-se dos demais métodos ágeis por dar maior ênfase ao gerenciamento do projeto. Reúne, entre outras, 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). Todo projeto Scrum normalmente inicia-se com uma visão geral do produto, que deve
fornecer 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).

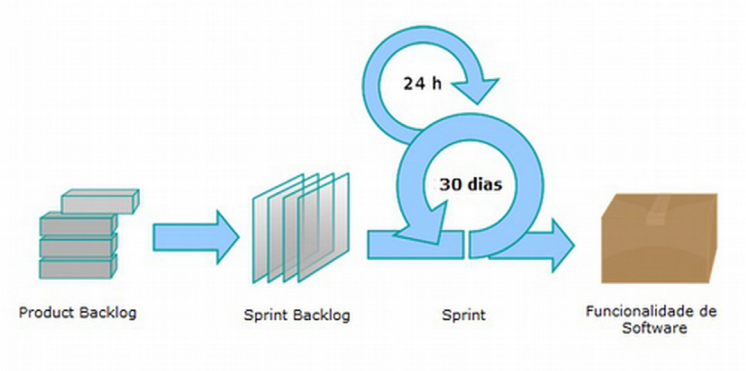


Figura 3: Representação geral do funcionamento Scrum

Fonte: Greoffe (2012).

Durante a fase de desenvolvimento do sistema foram realizadas muitas iterações (sprints), nas quais, em cada uma delas, procurava-se desenvolver e corrigir pequenas partes do sistema, sendo esses testados e integrados no final, procurando satisfazer um subconjunto de requisitos do projeto. O sistema encontra-se agora em fase de testes pelos usuários em que, por meio de interações dire-tas com os usuários, 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 e Oliveira (2015).

Apresentando o Selenium e TestNG

O alto interesse e a popularidade da internet produziram um aumento significativo na procura por aplicativos e sistemas da web. Os usuários esperam que esses sistemas sejam cada vez mais está-veis, confiáveis, seguros e compatíveis com os navegadores que estão utilizando. O procedimento que se concentra no teste de um sistema web é chamado de teste web, e é uma ótima maneira de se obter um bom nível de qualidade nesses sistemas (Vila et al., 2017). Por causa da crescente necessidade de rapidez no desenvolvimento de sistemas, está cada vez mais difícil realizar testes seguros e confiáveis para esses softwares. Em tais situações, o uso de testes automatizados é a melhor solução. (Vila et al., 2017).

[7] Disponível em: <https://netbeans.org/>.

Os testes automatizados usam softwares especializados para controlar a execução de algoritmos de teste e comparar os resultados reais com resultados previstos (Vila et al., 2017). As principais vantagens dos testes automatizados são (Vila et al., 2017):

- Eficiência – O tempo para execução de *scripts* de teste automatizados é menor do que executá-los manualmente, pois são executados por ferramentas que facilitam esse teste.
- Repetibilidade – A mesma lista de casos de teste pode ser executada novamente de maneira absolutamente idêntica à anterior, o que elimina o risco de erro ao recriar esse teste se fosse feito manualmente.
- Escopo do teste – Torna-se mais fácil rastrear a quantidade de código que foi coberta pelos casos de teste, e se há funcionalidades importantes que ainda não foram testadas.

Dentre os *frameworks* especializados em testes automatizados para web, o Selenium talvez seja o mais conhecido. O Selenium é composto de um conjunto de diferentes ferramentas de software, cada uma com uma abordagem diferente para suportar a automação de testes de sistemas web de todos os tipos. Esse conjunto de ferramentas permite localizar elementos da interface do usuário e comparar os resultados esperados do teste com o comportamento real do aplicativo. Um dos principais recursos do Selenium é o suporte para a execução de testes em vários navegadores. (The Selenium…, 2020).

O Selenium surgiu pela primeira vez em 2004, quando Jason Huggins estava testando uma aplicação web que havia desenvolvido e percebeu que poderia usar melhor seu tempo do que realizar manualmente os mesmos testes a cada mudança que implementava em sua aplicação. Assim, Huggins desenvolveu uma biblioteca Javascript que poderia realizar interações com as páginas de seu software web, permitindo a execução automática dos testes em vários navegadores. Essa biblioteca acabou se tornando o Selenium Core, que sustentava todas as funcionalidades do Selenium Remote Control (RC) e do Selenium IDE. O Selenium RC foi inovador porque nenhum outro produto permitia que alguém controlasse um navegador de sua escolha antes dele. (The Selenium…, 2020)

Por outro lado, o Selenium ainda possuía algumas limitações. Devido ao seu mecanismo de automação baseado em Javascript, a maioria dos navegadores já não executavam testes mais sofisticados, pois esses navegadores iniciaram aplicar limitações de segurança em relação ao Javascript. Além disso, as aplicações web tornavam-se cada vez mais sofisticadas, usando todos os tipos de recursos especiais que os novos navegadores ofereciam, tornando as restrições cada vez mais complexas. Dessa forma, no ano de 2006, o engenheiro Simon Stewart, do Google, começou a trabalhar em um projeto denominado WebDriver, que era uma ferramenta de teste que “conversava” diretamente com o navegador. Para tanto, o WebDriver usava um método “nativo” para controlar tanto o navegador quanto seu sistema operacional, evitando assim as restrições de um ambiente de JavaScript. (The Selenium…, 2020)

Por fim, no ano de 2008 ocorreu a fusão do pacote Selenium com o WebDriver. Por um lado, o Selenium tinha enorme apoio comunitário e comercial, mas o WebDriver era claramente a ferramenta do futuro. A junção das duas ferramentas forneceu um conjunto comum de recursos para todos os usuários e trouxe algumas das mentes mais brilhantes da automação de testes sob o mesmo teto de desenvolvimento. (The Selenium…, 2020)

Entretanto, o Selenium WebDriver não possui funcionalidade de agrupamento e ordenação dos testes automatizados. Essa limitação foi evitada, nesse projeto, utilizando a biblioteca TestNG. TestNG[8] é uma estrutura de teste projetada para superar as limitações da estrutura de teste Junit, e fornece algumas novas funcionalidades que o tornam mais poderoso que o Junit. O TestNG abrange todas as categorias de testes, como testes unitários, funcionais e de integração. (Gojare et al., 2015).

[8] Disponível em: <https://testng.org>.

Criando um teste automatizado no sistema BDPFG utilizando Selenium

Considerando o fato de o sistema BDPFG ser um sistema web, foi necessário pensar numa maneira de implementar os testes automatizados para esse tipo de software. Ou seja, testes que fossem capazes de lidar com diversos tipos de navegadores e com as interfaces em HTML e JavaScript, além de tratar latência da rede, tempo de resposta do SGBD, etc.

Codificar esses tipos de teste exige mais do que chamadas de métodos e procedimentos. Para se testar uma funcionalidade é necessário a simulação das ações de um usuário interagindo com o programa, isto é, um clique do mouse, uma tecla pressionada, uma opção selecionada, entre outras ações. Dessa forma, escrever testes que verifiquem adequadamente todos estes componentes é uma tarefa não-trivial que exige um bom conhecimento e experiência.

Para tanto, como linguagem de programação, utilizou-se o Java, e como ambiente de desenvolvimento, o NetBeans. Essas ferramentas foram as mesmas utilizadas para o desenvolvimento do sistema BDPFG, o que facilitou a implementação dos testes. Como *framework* de testes, foi escolhido o Selenium, que oferece bibliotecas próprias para a codificação desses testes automatizados. Também foi selecionada a biblioteca TestNG para possibilitar a ordenação e configuração de dependências dos testes feitos com Selenium.

O projeto de testes do sistema BDPFG foi desenvolvido utilizando-se o gerenciador de projetos Maven. 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) (The Apache Software Foundation, 2020).

Todos os projetos desenvolvidos com 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, entre outros. O Maven faz o *download* de bibliotecas Java e seus *plug-ins* dinamicamente de um ou mais repositórios e armazena-os em uma área de *cache* local. (The Apache Software Foundation, 2020). A Figura 4 ilustra uma parte do arquivo pom.xml utilizado no projeto de testes do sistema BDPFG:

```
<name>bdgfTests</name>
<dependencies>
    <!-- https://mvnrepository.com/artifact/org.seleniumhq.seleniun
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-api</artifactId>
        <version>3.141.59</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.seleniumhq.seleniun
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-remote-driver</artifactId>
        <version>3.141.59</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.seleniumhq.seleniun
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-firefox-driver</artifactId>
        <version>3.141.59</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.seleniumhq.seleniun
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-support</artifactId>
        <version>3.141.59</version>
    </dependency>
```

Figura 4. Parte do arquivo pom.xml do projeto de testes.

O projeto de testes automatizados do sistema BDPFG segue a arquitetura de duas camadas:

- Camada de cliente – contém todos os testes automatizados funcionais, criados e executados pelos usuários do projeto.
- Camada de negócios – contém toda a lógica de negócios da aplicação dos testes (BDPFG), suas páginas e principais funcionalidades. Além disso, contém as classes que definem os dados a serem utilizados nos testes e como serão utilizados, bem como métodos para definição de navegador e verificação do sistema operacional, limpeza do banco de dados, etc.

Essas camadas citadas anteriormente estão divididas em pacotes no projeto Java, os quais facilitam a manutenção do código e colocam organização aos testes. Os pacotes disponíveis no projeto são:

- **br.embrapa.cnptia.bdgf.core** – este pacote contém funcionalidades fundamentais e que serão utilizadas em todo projeto, como definição da plataforma, o navegador que executará os testes e seu *driver*, o usuário que fará login no sistema BDPFG, entre outros
- **br.embrapa.cnptia.bdgf.dataprovider** – neste pacote (e suas subdivisões) são definidas as classes de leitura de arquivos tabulares (no formato CSV e TSV). Essas classes definem os caminhos onde os arquivos estão, quais linhas e colunas serão lidas e valores esperados nos resultados dos testes, entre outros parâmetros.
- **br.embrapa.cnptia.bdgf.pages** – este pacote (e suas subdivisões) contém a identificação das páginas do sistema BDPFG. Os componentes HTML de cada página são configurados em cada uma das classes que representa cada página. Além disso, métodos que representam as ações que podem ser executadas em cada página são implementados nestas classes (ex.: clique do mouse em botões, menus, etc.).
- **br.embrapa.cnptia.bdgf.tests** – o pacote (e suas subdivisões) contém os testes de fato. Nos testes estão incluídos os métodos e funções das classes do pacote “br.embrapa.cnptia.bdgf.pages”. Nos métodos de teste há a descrição dos dados a serem utilizados (do pacote “br.embrapa.cnptia.bdgf.dataprovider”), bem como o chamamento dos métodos definidos na definição das páginas.
- **br.embrapa.cnptia.bdgf.jdbc** – este pacote possui uma classe que acessa o banco de dados do sistema BDPFG por meio da biblioteca JDBC e retira registros antigos do banco, bem como cria registros necessários (se inexistentes) para o início e prosseguimento dos testes.
- **br.embrapa.cnptia.bdgf.utils** – o pacote possui classes com diversas utilidades, tais como manipulação de arquivos JSON, formatação de números, diretório de *download* de arquivos e outras constantes utilizadas no projeto de testes.
- **br.embrapa.cnptia.bdgf.model** – este pacote contêm classes bases que tratam de páginas que contém visualização de dados em grade, *download* e *upload* de arquivos.
- **br.embrapa.cnptia.bdgf.primeui** – as classes deste pacote codificam o comportamento dos diversos componentes do *framework* Primefaces (https://www.primefaces.org/), o qual é utilizado no sistema BDPFG. Alguns desses componentes são: DataTable, Dialog e SelectCheckBoxMenu.

A Figura 5 ilustra a estrutura em blocos desses pacotes criados para o projeto de testes:

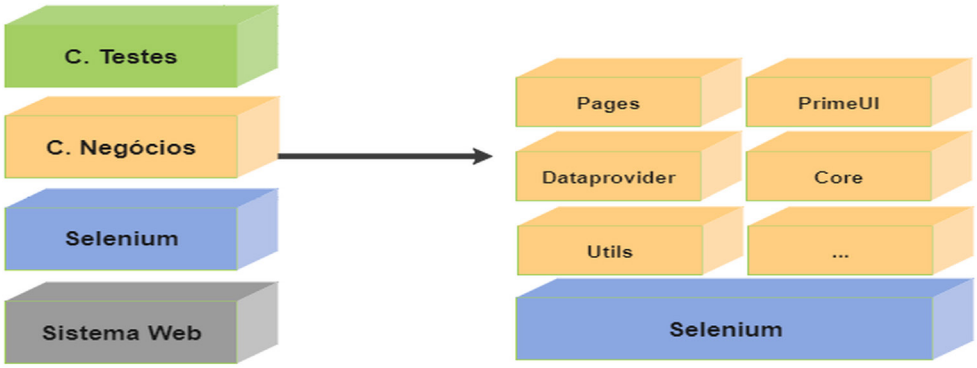
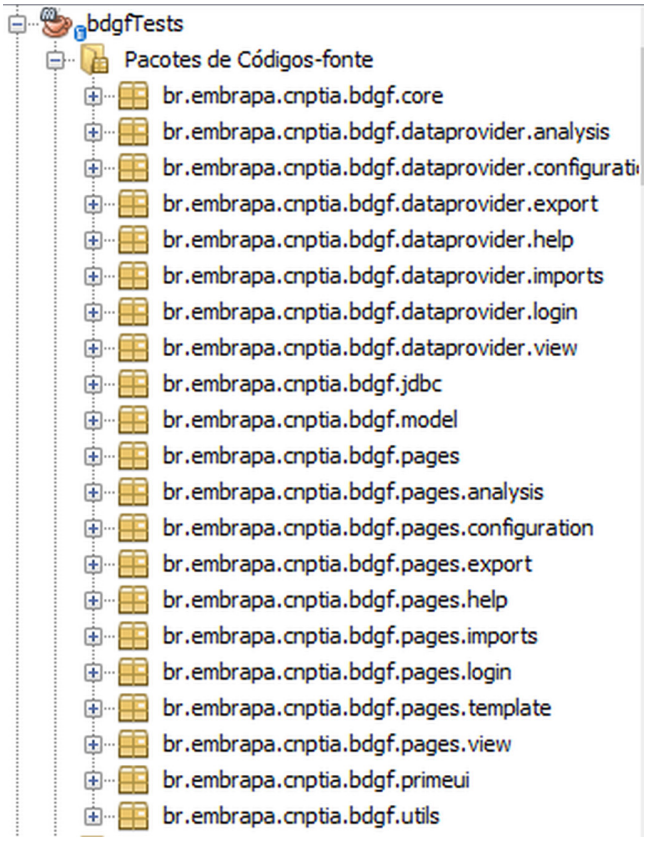


Figura 5: Estrutura em blocos do projeto de testes do sistema BDPFG.

Essa estrutura de projeto de testes é semelhante ao utilizado em Vila et al. (2017) e Gojare et al. (2015), os quais relatam resultados muito satisfatórios na implementação e manutenção dos algoritmos com essa forma de estruturação de código. As Figuras 6 e 7 a seguir ilustram os pacotes descritos anteriormente no ambiente do NetBeans, que foi utilizado para implementar o projeto de testes do sistema BDPFG.



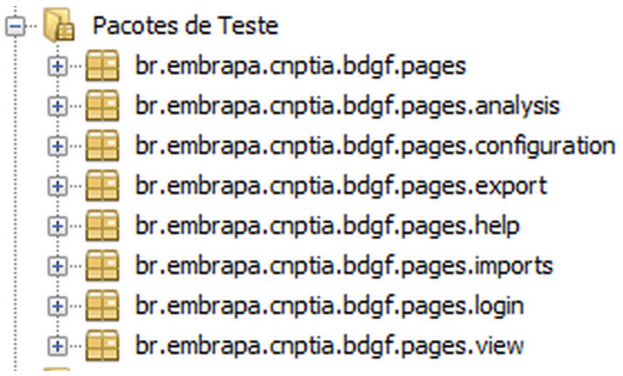


Figura 6: Pacotes que representam a lógica de negócio dos testes em relação ao sistema BDPFG..

Há ainda os arquivos tabulares utilizados para execução dos testes automatizados (Figura 7), que são mapeados pelas classes contidas no pacote “br.embrapa.cnptia.bdgf.dataprovider”. Esses arquivos ficam no diretório de recursos do projeto, que ainda conta com as últimas versões dos *driver*s de acesso ao navegador Firefox (geckodriver):

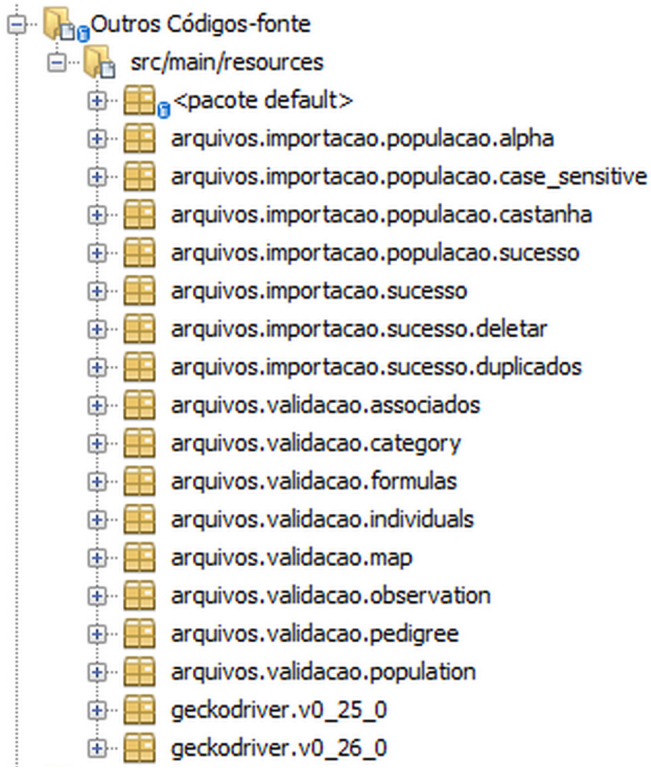


Figura 7: Pacotes onde estão localizados os testes propriamente ditos, representando a camada de cliente de execução dos testes.

Com essa distribuição de pacotes, a organização e implementação de novos testes automatizados ficou mais rápido e eficiente, pois classes com métodos e variáveis utilizados globalmente ficaram isoladas em alguns pacotes e as classes de páginas do sistema a serem testadas ficaram localizadas no pacote “br.embrapa.cnptia.bdgf.pages” e suas subdivisões. Um exemplo de classe global é a classe de configuração de login, que está localizada no pacote “br.embrapa.cnptia.bdgf.core”. Parte de seu conteúdo é exibida na Figura 8:

```
public class LoginConfig {

    private final static String DEFAULT_GROUP = "selenium";

    private final static String TEST_GROUP = System.getProperty("test.group");

    private final static String BASE_URL_USER = System.getProperty("base.url.user");

    private final static String BASE_URL_PASSWORD = System.getProperty("base.url.password");

    public static String getGroup() {
        return Optional.ofNullable(TEST_GROUP).
                filter(e -> e != null && !e.trim().equals("") && e.length() != 0).
                isPresent() ? TEST_GROUP : DEFAULT_GROUP;
    }

    public static String getDownloadDirectory() {
        String _downDir = Optional
                .ofNullable(System.getProperty("browser.download.dir"))
                .filter(e -> e != null && !e.trim().equals("") && e.length() != 0).
                isPresent() ? System.getProperty("browser.download.dir") : System.getProperty("

        return _downDir.concat("/").concat(getGroup());
    }

    public static String getUser() {
        return Optional.of(BASE_URL_USER).
                filter(e -> e != null && !e.trim().equals("") && e.length() != 0).
                get();
    }
```

Figura 8: Classe de configuração de login para o projeto de testes do sistema BDPFG.

Outra classe importante desse pacote é a que define a localização e inicialização do *driver* de uso do navegador Firefox (geckodriver). Parte de seu conteúdo é exibida na Figura 9:

```
public class DriverFactory {

    private static WebDriver webDriver;

    private static DriverFactory driverFactory;

    private final static boolean BROWSER_START_MAXIMIZED = Boolean.getBoolean("browser.start.

    private final static long SELENIUM_DEFAULT_IMPLICITLY_WAIT = getLong(System.getProperty("

    private final static boolean HEADLESS = Boolean.getBoolean("browser.headless");

    private final static String BASE_DIR_GECKODRIVER = System.getProperty("user.dir") + "/src/

    public final static String FIREFOX_LOG_FILE = "firefox.log";

    private DriverFactory() {
        init();
    }

    public static DriverFactory newInstance() {
        if (null == driverFactory) {
            driverFactory = new DriverFactory();
        }
        return driverFactory;
    }

    public WebDriver getDriver() {
        return webDriver;
    }
```

Figura 9: Classe de configuração e inicialização do *driver* do navegador Firefox.

Há também uma classe que realiza a limpeza inicial do banco de dados de testes do BDPFG, e que fica localizada no pacote “br.embrapa.cnptia.bdgf.jdbc”. Essa classe utiliza o *driver* JDBC[9] para se conectar ao banco do siste-ma. A Figura 10 mostra parte dessa classe:

```
public class JdbcBDGF {

    private final static String POSTGRES_URL = System.getProperty("postgres.url");

    private final static String POSTGRES_USER = System.getProperty("postgres.user");

    private final static String POSTGRES_PASSWORD = System.getProperty("postgres.password");

    public Connection conecta() throws SQLException {
        return DriverManager.getConnection(POSTGRES_URL, POSTGRES_USER, POSTGRES_PASSWORD);
    }

    /**
     * Remover dados para dos testes de ViewIndividuals
     */
    public void deleteAllTestData() {
        Long _groupid = getGroupID(LoginConfig.getGroup());

        deleteLogByGroupid(_groupid);
        deleteContemporaryByGroupid(_groupid);
        deleteFormularByGroupid(_groupid);
        deleteCategoryByGroupid(_groupid);
        deleteObservationsByGroupid(_groupid);

        //removendo genomas
        GenomeDataProvider dataProvider = new GenomeDataProvider();
        Arrays.asList(dataProvider.adicionarGenoma()).forEach(e -> deleteGenoma(e[0].toString(), _groupid));
        Arrays.asList(dataProvider.editarGenoma()).forEach(e -> deleteGenoma(e[1].toString(), _groupid));

        deleteIndividualPopulationByGroupid(_groupid);
        deletePopulationByGroupid(_groupid);
```

Figura 10: Classe de conexão e manipulação inicial do banco de dados BDPFG.

[9] Disponível em: <https://www.oracle.com/br/database/technologies/appdev/jdbc.html>.

Existe mais um grupo de classes fundamentais para o projeto, e que está localizado no pacote “br.embrapa.cnptia.bdgf.primeui”. Essas classes manipulam os componentes do *framework* PrimeFaces, que são componentes HTML com uma melhor qualidade visual, com mais recursos funcionais e baseados na biblioteca JavaServer Faces[10]. Esses componentes são utilizados no sistema BDPFG. Um exemplo de classe de manipulação desses componentes no projeto de testes é mostrado na Figura 11, a qual implementa um algoritmo para tratar o componente DataTable, responsável por exibir os dados em formato de grade.

```
public class LoginPage extends BasePage {

    @FindBy(how = How.XPATH, using = "//input[@id='formLogin:email']")
    private WebElement inputTextEmail;

    @FindBy(how = How.XPATH, using = "//input[@id='formLogin:password']")
    private WebElement inputTextPassword;

    @FindBy(how = How.XPATH, using = "//button[@id='formLogin:btnLogin']")
    private WebElement commandButtonLogin;

    @FindBy(how = How.XPATH, using = "//*[@id='menuForm:menuPrincipal']")
    private WebElement menuTemplate;

    public LoginPage(WebDriver driver) {
        super(driver);
    }

    /**
     *
     * @param email
     * @param password
     */
    public void login(String email, String password) {
        sendKeysOf(inputTextEmail, email);
        sendKeysOf(inputTextPassword, password);
        click(commandButtonLogin);
    }
```

Figura 11: Classe que manipula o componente DataTable no projeto de testes.

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

Diversos outros componentes são mapeados por classes nesse pacote, o que facilita na implementação das páginas do sistema BDPFG, que estão localizados no pacote “br.embrapa.cnptia.bdgf.pages”. Um desses componentes é utilizado frequentemente no sistema, que é o faz o upload de arquivos. Abaixo (Figura 12) é exibida parte dessa classe:

```
public class FileUpload extends AbstractUI {

    /**
     *
     * @param element
     * @param webDriver
     */
    public FileUpload(WebElement element, WebDriver webDriver) {
        this.driver = webDriver;
        this.element = element;
        this.elementId = element.getAttribute("id");
    }

    /**
     *
     * @param localFile
     */
    public void uploadFile(String localFile) {
        element = waitToBeClickable(element);
        WebElement _fileUpload = element.findElement(By.xpath("//input[@id='" + elementId +
        _fileUpload.sendKeys(localFile);
    }
}
```

Figura 12: Classe que mapeia componente responsável por upload de arquivos.

Dentro do pacote mais importante do projeto (“br.embrapa.cnptia.bdgf.pages”), as páginas relativas ao sistema BDPFG são codificadas por meio de diversas classes organizadas em diversos “subpacotes”, o que facilita a manutenção da implementação dos testes automatizados. Duas classes bases para essas páginas estão localizadas no pacote raiz “br.embrapa.cnptia.bdgf.pages”. Essas classes são a BaseWait e BasePage. A primeira implementa, entre outras coisas, os tempos de espera de resposta dos componentes do BDPFG. A segunda estende a primeira e inicia o *driver* geckodriver para as páginas, além de criar uma instância do objeto PageFactory, que é usado para inicialização de objetos (componentes HTML) das páginas sem ter que usar apenas o método FindElement dentro de cada método, permitindo o uso da anotação @FindBy para encontrar todo WebElement da página. As Figuras 13 e 14 apresentam o código das classes BaseWait e BasePage, respectivamente:

```
public abstract class BaseWait {

    protected WebDriver driver;

    //TODO: checar se vai ser colocado no pom.xml
    private static final long REDRAW_UI_ELEMENTS_TIMEOUT_SEC = 300;

    private static final long POLLING_INTERVAL_IN_MILLIS = 200;

    public BaseWait() {
    }

    private Wait<WebDriver> baseVisible(long waitTimeOutInSeconds) {
        return new FluentWait<>(driver)
                .withTimeout(Duration.ofSeconds(waitTimeOutInSeconds))
                .pollingEvery(Duration.ofMillis(POLLING_INTERVAL_IN_MILLIS))
                .ignoring(NoSuchElementException.class, StaleElementReferenceException.cla
    }

    protected void waitVisibilityOfBlockUI() {
        waitVisibilityOf(driver.findElement(By.className("ui-blockui")));
    }

    protected void waitInvisibilityOfBlockUI() {
        waitInvisibilityOf(driver.findElement(By.className("ui-blockui")));
    }

    protected void waitInvisibilityOfBlockUI(long timeout) {
        waitInvisibilityOf(driver.findElement(By.className("ui-blockui")), timeout);
    }
```

Figura 13: Classe BaseWait presente do pacote de páginas.

```java
public abstract class BasePage extends BaseWait {

    /**
     *
     * @param driver
     */
    public BasePage(WebDriver driver) {
        this.driver = driver;
        PageFactory.initElements(driver, this);
    }

    /**
     *
     * @return
     */
    public String getURL() {
        return driver.getCurrentUrl();
    }
}
```

Figura 14: Classe BasePage presente no pacote de páginas e que estende BaseWait.

Um exemplo de página implementada nesse pacote é a de visualização de indivíduos do BDPFG, uma das mais importantes funcionalidades do sistema. A classe que referencia essa página tem muitas variáveis e métodos implementados, e se chama IndividualsPage, presente em “br.embrapa.cnptia.bdgf.pages.view”. A Figura 15 mostra apenas uma parte dela:

```java
public long countTotalPorPopulacao(IndividualPopulation population) {

    checkboxMenuPopulation.uncheckAll();
    checkboxMenuPopulation.check(population.getPopulation());

    WebElement _element = dataTableViewIndividuals.getTotal();
    Log.debug("_element=[ " + _element + " ]");

    if (_element == null) {
        Log.debug("_element=[ " + _element + " ] não encontrado");
        return 0;
    }

    String elementTotal = _element.getText();
    String result = elementTotal.substring(elementTotal.lastIndexOf(" ") + 1);

    Log.debug("Parsing String to Long =[ " + elementTotal.substring(elementTotal.lastIndexOf("
    return NumberUtils.getLong(result);
}

public boolean deletarIndividual(String nomeIndividuo) {
    waitInvisibilityOfBlockUI();
    dataTableViewIndividuals.columnFilter(NOME, nomeIndividuo);
    waitInvisibilityOfBlockUI();

    if (dataTableViewIndividuals.countRow() > 1) {
        Log.error("Falha ao buscar animais.");
        return false;
    }

    dataTableViewIndividuals.click(NOME, nomeIndividuo, ViewIndividualsDataTable.FixedColumns.
    click(deleteIndividualButton);
```

Figura 15: Parte do código da classe IndividualsPage.

As muitas funcionalidades da tela de visualização de indivíduos são implementadas nesta classe, como a função de deleção de indivíduo (deletarIndividual), vista na figura anterior. Os componentes também já são iniciados no início da classe com o recurso do PageFactory (com a anotação @FindBy), discutido anteriormente.
Outro exemplo de classe de página implementada é a LoginPage, onde se pode observar o recurso do PageFactory implementado com o uso da anotação @FindBy (Figura 16):

```
public class LoginPage extends BasePage {

    @FindBy(how = How.XPATH, using = "//input[@id='formLogin:email']")
    private WebElement inputTextEmail;

    @FindBy(how = How.XPATH, using = "//input[@id='formLogin:password']")
    private WebElement inputTextPassword;

    @FindBy(how = How.XPATH, using = "//button[@id='formLogin:btnLogin']")
    private WebElement commandButtonLogin;

    @FindBy(how = How.XPATH, using = "//*[@id='menuForm:menuPrincipal']")
    private WebElement menuTemplate;

    public LoginPage(WebDriver driver) {
        super(driver);
    }

    /**
     *
     * @param email
     * @param password
     */
    public void login(String email, String password) {
        sendKeysOf(inputTextEmail, email);
        sendKeysOf(inputTextPassword, password);
        click(commandButtonLogin);
    }
```

Figura 16: Parte do código da classe LoginPage.

As classes que mapeiam os arquivos de dados utilizados nos testes estão localizadas em “br.embrapa.cnptia.bdgf.dataprovider”. Um exemplo dessa classe de dados é a utilizada no teste de visualizar indivíduos, mostrada anteriormente. Essa classe está implementada em “br.embrapa.cnptia.bdgf.dataprovider.view”, e pode ser vista Figura 17:

```
public class IndividualsDataProvider {

    @DataProvider(name = "baixarArquivoIndividuos")
    public Object[][] baixarArquivoIndividuos() {

        // {nome da populacao do arquivo, colunas para desmarcar, caminho do arquivo baixado,
        return new Object[][]{
            {"SELENIUM.ALPHA", "ORIGINALID,FATHER,FATHERID,MOTHER,MOTHERID,INSERT DATE,START
            {"SELENIUM.ALPHA", "INDIVIDUALID,FATHER,FATHERID,MOTHER,MOTHERID,INSERT DATE,STAR
        };
    }

    @DataProvider(name = "baixarArquivoObservations")
    public Object[][] baixarArquivoObservations() {

        // {nome da populacao do arquivo, colunas para desmarcar, caminho do arquivo baixado,
        return new Object[][]{
            {"SELENIUM.ALPHA", "ORIGINALID,FATHER,FATHERID,MOTHER,MOTHERID,INSERT DATE,START
            {"SELENIUM.ALPHA", "INDIVIDUALID,FATHER,FATHERID,MOTHER,MOTHERID,INSERT DATE,STAR
        };
    }

    @DataProvider(name = "baixarArquivoPedigree")
    public Object[][] baixarArquivoPedigree() {

        // {nome da populacao do arquivo, colunas para desmarcar, caminho do arquivo baixado,
        return new Object[][]{
            {"SELENIUM.ALPHA", "pedigree.csv", "pediree-oririnalid-viewindividuals.csv", 42}
        };
    }
```

Figura 17: Classe IndividualsDataProvider que indica dados a serem utilizados nos teste de visualização de indivíduos.

Como pode ser visto na Figura 17, cada método dessa classe retorna um objeto com vários parâmetros indicando onde o arquivo de dados está localizado, variáveis utilizadas, retorno esperado do teste, entre outros. Cada método foi nomeado de forma a ter o mesmo nome do método de teste imple-mentado nos pacotes de teste.

Por fim, os testes propriamente ditos estão codificados dentro do pacote “br.embrapa.cnptia.bdgf.tests” e seus “subpacotes”. No projeto de testes desenvolvido, praticamente todas as funcionalidades do sistema BDPFG estão cobertas pelos testes. Sempre que há uma alteração ou correção no BDPFG, os testes devem ser verificados para se há necessidade de adequação do código. Entretanto, a estrutura do projeto facilita e muito essa manutenção por qualquer desenvolvedor da equipe.

Da mesma forma que existe uma classe BasePage na camada de negócios do projeto de testes, há a classe BaseTest na camada cliente (de testes). Essa classe é estendida na maioria dos testes do sistema, sendo que ela faz o carregamento do geckodriver em memória, faz o login do sistema BDPFG, além de definir a linguagem padrão (atualmente, português do Brasil, com sigla PT-BR) e grupo utilizado para teste (um nome de grupo para organização dos dados definido pelo usuário). A Figura 18 exibe parte do código dessa classe:

```
public BaseTest() {
}

@BeforeClass(alwaysRun = true)
public void setUpClass() throws Exception {
    try {
        if (driverFactory == null) {
            driverFactory = DriverFactory.newInstance();
            driverFactory.open(BASE_URL);
            new LoginPage(driverFactory.getDriver()).login(LoginConfig.getUser(), Lo
            templatePage = new TemplatePage(driverFactory.getDriver());

            String _group = Optional.ofNullable(TEST_GROUP).filter(_value -> !_value
            templatePage.selectLanguage(Language.PT_BR);
            templatePage.selectGroup(_group);
        }

    } catch (Exception ignone) {
        driverFactory.quit();
        throw new SkipException("desired Message");
    }
}

@AfterMethod(alwaysRun = true)
public void tearDownMethod() throws Exception {
    driverFactory.getDriver().navigate().to(BASE_URL);
}

@AfterSuite(alwaysRun = true)
public void tearDownSuite() throws Exception {
    driverFactory.quit();
}
```

Figura 18: Classe BaseTest utilizada como base em outros testes do projeto.

Um dos testes de maior destaque é o que testa a visualização de indivíduos. Esse teste, assim como o restante, utiliza-se de uma classe (IndividualsPage, neste caso) criada no pacote “br.embrapa.cnptia.bdgf.pages” para chamar os métodos relativos às funcionalidades dessa página do sistema. A Figura 19 exibe parte do código dessa classe de teste:

```
public class ViewIndividualsTest extends BaseTest {

    public ViewIndividualsTest() {
    }

    @BeforeMethod(alwaysRun = true)
    public void setUpMethod(Method method) {
        Log.methodInfo(method);
        templatePage.selectMenu(MenuPrincipal.Menu.VIEW, MenuPrincipal.SubMenu.VIEW_INDIVIDUALS);
    }

    @Test(testName = "checarIndividualDesconhecido", description = "Checando se o individual DESCONHEC
            dataProvider = "checaOriginalIdDeconhecido", dataProviderClass = GroupsDataProvider.class)
    public void checarIndividualDesconhecido(String group, String originalid, String populacao, String
        boolean sucess = new IndividualsPage(driverFactory.getDriver()).hasOriginalID(originalid, popu
        // é checado na lista e se a mensagem apareceu do growl é de sucesso
        Assert.assertTrue(sucess);
    }

    @Test(testName = "baixarArquivoIndividuos", description = "Testa o download de arquivo individuos"
            dataProvider = "baixarArquivoIndividuos", dataProviderClass = IndividualsDataProvider.clas
    public void baixarArquivoIndividuos(String nomePopulacao, String colunasDesmarcar, String currentN
        long totalLines = new IndividualsPage(driverFactory.getDriver()).downloadIndividual(currentNam
        Assert.assertEquals(expectedLines, totalLines);
    }

    @Test(testName = "baixarArquivoObservations", description = "Testa o download de arquivo obsevatio
            dataProvider = "baixarArquivoObservations", dataProviderClass = IndividualsDataProvider.cl
    public void baixarArquivoObservations(String nomePopulacao, String colunasDesmarcar, String curren
        long totalLines = new IndividualsPage(driverFactory.getDriver()).downloadObservation(currentNa
        Assert.assertEquals(expectedLines, totalLines);
    }
```

Figura 19: Classe que implementa os testes da página de visualização de indivíduos.

Observa-se, pela Figura 19, o uso das classes de dados (presentes nos pacotes dentro de “br.embrapa. cnptia.bdgf.dataprovider”) em cada método de teste, representadas pela anotação @Test. A defi-nição e configuração da classe de dados deve ser escrita dentro dos parâmetros dessa anotação.

Outro teste de grande relevância é o de importação de dados de indivíduos, pois esta é uma funcionalidade de muita utilização no sistema BDPFG, a qual faz diversas verificações durante esse processo, como a verificação de animais duplicados, formato do arquivo de entrada, dados faltantes,

```
@Test(testName = "importarIndividualComSucesso", description = "Importacao de individuao com
        dataProvider = "importacaoIndividualSucesso", dataProviderClass = IndividualDataProv
public void importarIndividualComSucesso(String description, String filePath) {
    //pafina de importacao
    IndividualsPage importIndividual = new IndividualsPage(driverFactory.getDriver());

    //checando arquivo
    Status status = importIndividual.chose(filePath);
    AssertJUnit.assertEquals(Status.SUCESSO_INSERT_OU_UPDATE, status);

    //importando arquivo
    boolean sucess = importIndividual.submit();
    Assert.assertTrue(sucess);
}

@Test(testName = "importarIndividualComErro", description = "Validando a importacao com prob
        dataProvider = "importacaoIndividualErros", dataProviderClass = IndividualDataProvid
public void importarIndividualComErro(String description, String filePath, String growlMessa
    //pafina de importacao
    IndividualsPage importIndividual = new IndividualsPage(driverFactory.getDriver());
    //checando arquivo
    String statusMessage = importIndividual.checkFileWithError(filePath).replace("\n", "").r
    AssertJUnit.assertEquals(growlMessage, statusMessage);
}

@Test(testName = "baixarArquivoDeErros", description = "Faz o download do arquivo de erros",
        dataProvider = "baixarArquivoDeErros", dataProviderClass = IndividualDataProvider.cl
public void baixarArquivoDeErros(String filePath, String expectedMessageGrowlError, String n
        String novoNomeArquivoBaixado, long expectedLines) {
    // faz a importação do arquivo
    IndividualsPage importIndividual = new IndividualsPage(driverFactory.getDriver());
    String statusMessage = importIndividual.checkFileWithError(filePath);
```

Figura 20: Classe que implementa testes na página de importação de indivíduos.

A configuração e ordenação dos testes foi realizada com o uso da biblioteca TestNG, que permite ordená-los de forma que um teste que dependa da execução de outro (ou outros) não seja realizado antes. O arquivo testng.xml é onde deve-se configurar essa ordenação, como pode ser visto na Figura 21:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite
        name="Suite - Banco de Dados de Genótipos e Fenótipos Testes"
        allow-return-values="true">
        <test name="all-tests" preserve-order="true">
                <classes>
                        <class name="br.embrapa.cnptia.bdgf.pages.login.LoginTest">
                                <methods>
                                        <!--<include name="loginComErro" />-->
                                        <include name="loginComSucesso" />
                                </methods>
                        </class>

                        <class name="br.embrapa.cnptia.bdgf.pages.configuration.Populatic
                                <methods>
                                        <include name="importarPopulacaoSucesso" />
                                        <include name="exportarArquivoPopulacao" />
                                        <include name="inserirPopulacao" />
                                        <include name="editarPopulacaoExiste" />
                                        <include name="editarPopulacao" />
                                        <include name="importarPopulacaoComErro" />
                                        <include name="deletePopulacao" />
                                </methods>
                        </class>

                        <class name="br.embrapa.cnptia.bdgf.pages.imports.IndividualsTest
                                <methods>
                                        <include name="importarIndividualComSucesso" />
                                        <include name="importarIndividualComErro" />
                                        <include name="baixarArquivoDeErros" />
                                </methods>
```

Figura 21: Arquivo XML da biblioteca TestNG de configuração de ordenação dos testes.

Observa-se, pelo arquivo da Figura 21, a cláusula preserve-order, a qual permite que os testes sejam executados na ordem que estão descritos no arquivo testng.xml. Além disso, cada método de teste está descrito dentro de cada classe de testes a qual pertence, facilitando a visualização e organização dos testes do projeto.

Resultados e Discussão

Inicialmente, a automação dos testes do sistema BDPFG por meio do *framework* Selenium teve um custo um pouco alto para a equipe do projeto em relação ao tempo, pois além da equipe de desenvolvimento ser pequena, a curva de aprendizagem do foi um pouco longa. Além disso, poucos da equipe tinham experiência com desenvolvimento de testes automatizados em Java. A manutenção, entretanto, ficou menos árdua, pois essa tinha maior correlação com alterações e correções realizadas no sistema BDPFG do que com novas necessidades de aprendizagem. Obviamente que

implementações e alterações mais complexas do sistema demandam um trabalho maior para adaptação dos testes.

A maioria dos testes implementados buscou identificar defeitos apenas nas funcionalidades disponibilizadas pela interface do sistema Web, como os testes para analisar a visualização de animais, por exemplo. Contudo, alguns desses testes tiveram um objetivo adicional, que foi identificar problemas de desempenho na carga de dados, como os testes que verificam e analisam a importação de animais, pedigree e fenótipos, e os testes relacionados a identificação de animais duplicados, cujo tempo de execução costuma ser alto. Observou-se, por diversas vezes, que os testes falhavam devido a demora dos servidores (tanto de homologação quanto de testes) em carregar tais dados, mostrando uma das importâncias desses testes. Com isso, melhorias foram (e continuam sendo) realizadas no sistema BDPFG e também no sistema gerenciador de banco de dados (PostgreSQL), procurando otimizar suas configurações e também as funções da linguagem procedural PL/pgSQL implementadas para o sistema.

No momento, com a atual cobertura de testes e de dados utilizados, a execução de todo o projeto está demorando cerca de duas horas e trinta minutos (Figura 22) no servidor GitLab[11] da Embrapa Informática Agropecuária. Em paralelo, a equipe está desenvolvendo uma imagem em Docker[12] com apenas os softwares necessários para execução do sistema BDPFG para verificar se o tempo de execução dos testes diminui num ambiente sem interferência de processos externos desnecessários ao funcionamento do sistema, e com isso também analisar o desempenho do próprio BDPFG nesse ambiente.

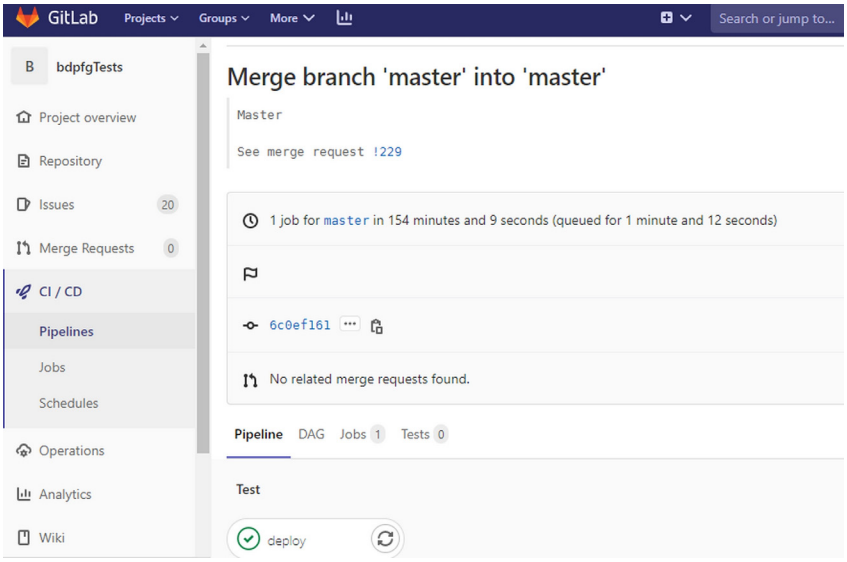


Figura 22. Resultado e tempo de execução dos testes no GitLab em um dos Pipelines.

Esse projeto de testes buscou cobrir praticamente toda a interface do sistema BDPFG, abrangendo quase que todas as ferramentas presentes no software, como visualização e importação de dados, cadastro de grupo e populações, *backup* de dados, visualização de *logs*, etc. Mesmo que a cobertura quase que integral do sistema não indique que a qualidade dos testes esteja adequada, a equipe procurou escrever testes com algoritmos de qualidade, e que diminuíssem a quantidade de erros em produção.

[11] Disponível em: <https://www.gitlab.cnptia.embrapa.br/>.

[12] Disponível em: <https://www.docker.com/>.

Muitos outros testes continuam sendo implementados nesse projeto de testes automatizados utilizando Selenium. O objetivo principal do trabalho foi mostrar uma visão geral de como foi construído a estrutura dos algoritmos de testes, ou seja, como foram organizados os pacotes de classes e as bibliotecas utilizadas, abrindo um caminho para que outros sistemas Web da Embrapa, que não utilizam nenhum tipo de teste automatizado, possa iniciar algo semelhante a partir de um já existente, e que está funcionando muito bem para a equipe de desenvolvimento. Outros trabalhos envolvendo esse tipo de estrutura, como Vila et al. (2017) e Gojare et al. (2015), relataram bons resultados na implementação e manutenção dos testes. O projeto de testes do BDPFG está armazenado no GitLab da Embrapa Informática Agropecuária, e só pode ser acessado por pessoas autorizadas.

Além disso, esse projeto faz parte do processo de integração contínua do sistema BDPFG por meio do recurso GitLab CI (*GitLab Continous Integration*). Dessa forma, toda vez que o código do sistema é modificado ou corrigido no GitLab, os testes automatizados são disparados de forma automática pelo servidor, avisando (via e-mail) toda a equipe de desenvolvedores sobre erros encontrados na construção e *deploy* do sistema.

Considerações Finais

O desenvolvimento de testes automatizados, principalmente para sistemas web, não é algo trivial, ainda mais quando o sistema a ser testado começa a ganhar muitas funcionalidades. Inicialmente, quando o sistema está com poucos módulos desenvolvidos, testes manuais garantem o software está sendo desenvolvido sem erros. Contudo, com o tempo e a evolução do sistema, os testes manuais podem incorrer em diversas falhas na identificação de inconsistências no sistema. Dessa forma, o desenvolvimento de testes automatizados se torna uma saída viável para esse dilema.

O projeto desenvolvido em Java proposto para testes automatizados do sistema Web BDPFG utilizando Selenium é uma excelente ferramenta para automatizar os processos manuais que já estavam sendo realizados. A estrutura do projeto de testes desenvolvido para o sistema BDPFG segue algumas das melhores práticas a fim de facilitar a automação dos testes. Seu uso economiza tempo do desenvolvedor, tanto na criação de novos testes quanto na correção de outros, pois sua organização facilita encontrar onde ocorreu alterações, seja no nome de um componente HTML ou na mudança de fluxo de ação do usuário. Espera-se que esse projeto possa ser útil para o desenvolvimento de testes automatizados para outros sistemas Web vigentes na Embrapa.

Referências

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

GOJARE, S.; JOSHI, R.; GAIGAWARE, D. Analysis and design of Selenium webdriver automation testing framework. **Procedia Computer Science**, v. 50, p. 341-346, 2015. DOI: 10.1016/j.procs.2015.04.038.

GREOFFE, R. J. **Desenvolvimento ágil com Scrum**: uma visão geral. 2012. Disponível em: <http://www.devmedia.com.br/desenvolvimento-agil-com-scrum-uma-visao-geral/26343>. Acesso em: 10 abr. 2020.

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: <https://www.infoteca.cnptia.embrapa.br/infoteca/bitstream/doc/1022327/1/Doc133.pdf>. Acesso em: 10 abr. 2020.

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

SOARES, L. **Backing beans em JSF**. 2014. Disponível em: <https://siteluisdev.wordpress.com/2014/05/29/jsf-parte-3-backing-beans>. Acesso em: 10 abr. 2020.

THE APACHE SOFTWARE FOUNDATION. **Apache Maven Project**. Disponível em: <https://maven.apache.org/>. Acesso em: 9 jul 2020.

THE SELENIUM browser automation Project. Disponível em: <https://www.selenium.dev/documentation/en/>. Acesso em: 10 abr. 2020.

VIEIRA, F. D. T**estes automatizados no sistema BDGF com Arquillian e Junit**. Campinas: Embrapa Informática Agropecuária, 2017. 40 p. il. (Embrapa informática Agropecuária. Documentos, 153). Disponível em: <https://www.infoteca.cnptia.embrapa.br/infoteca/bitstream/doc/1082036/1/Doc153FabioD.pdf>. Acesso em: 9 jul. 2020.

VILA, E.; NOVAKOVA, G.; TODOROVA, D. Automation testing framework for web applications with Selenium WebDriver: opportunities and threats. In: INTERNATIONAL CONFERENCE ON ADVANCES IN IMAGE PROCESSING, 2017, Bangkok. **Proceedings...** New York: ICPS, 2017. p. 144-150. DOI: 10.1145/3133264.3133300.

Embrapa
Informática Agropecuária

MINISTÉRIO DA
AGRICULTURA, PECUÁRIA
E ABASTECIMENTO

PÁTRIA AMADA
BRASIL
GOVERNO FEDERAL

CGPE 16362