https://eventos.utfpr.edu.br//sicite/sicite2020

Manutenção do aplicativo de coleta e análise de dados do MIP/MID

Maintenance of the MIP / MID data collection and analysis application

Bruno Oliveira Galvão de Arruda
bruno.ogarruda@gmail.com
Universidade Tecnológica Federal do Paraná, Cornélio Procópio, Paraná, Brasil

Gabriel Costa Silva
gabrielcosta@utfpr.edu.br
Universidade Tecnológica Federal do Paraná, Cornélio Procópio, Paraná, Brasil

RESUMO

O presente trabalho é vinculado ao projeto de inovação que desenvolveu um aplicativo para coleta e análise de dados da tecnologia de manejo integrado de pragas da cultura da soja. O projeto visa aumentar a eficiência na coleta e análise de dados por meio da automatização do processo. O presente trabalho melhora a qualidade do aplicativo por meio da criação de testes unitários. Esses testes são fundamentais para garantir que modificações não influenciem de forma negativa no funcionamento do aplicativo. O *framework* JUnit foi usado para desenvolvimento dos testes. Para desenvolver o presente trabalho, primeiro, o JUnit foi estudado. Em seguida, testes para as classes de entidade dos pacotes `base`, `mid`, `mip`, `pulverization`, `survey` foram desenvolvidos. No total, 365 testes unitários foram desenvolvidos, garantindo que novas funcionalidades não impactem de forma negativa no código já existente. Adicionalmente, os testes ajudaram a identificar falhas presentes na atual implementação.

PALAVRAS-CHAVE: Manutenção de software. Testes. Java (Linguagem de programação de computador).

ABSTRACT

The present work is linked to the innovation project that developed an application for data collection and analysis of integrated pest management technology for soybean crops. The project aims to increase efficiency in data collection and analysis by automating the process. The present work improves the quality of the application through the creation of unit tests. These tests are essential to ensure that modifications do not negatively influence the functioning of the application. The JUnit framework was used to develop the tests. To develop the present work, JUnit was first studied. Then, tests for the entity classes of the base, mid, mip, pulverization, survey packages were developed. In total, 365 unit tests were developed, ensuring that new features do not negatively impact the existing code. Additionally, the tests helped to identify flaws present in the current implementation.

KEYWORDS: Maintenance, Software. Testing. Java (Computer program language).

Recebido: 19 ago. 2020.
Aprovado: 01 out. 2020.

INTRODUÇÃO

Com o uso exagerado de inseticidas, as lavouras ficam mais vulneráveis ao avanço rápido de pragas e o surgimento de novas pragas, pois eles afetam a ação dos agentes de controle biológico natural. O Manejo Integrado de Pragas (MIP) é uma tecnologia que agrupa um conjunto de procedimentos ambientais e econômicos para o controle efetivo de pragas, reduzindo o uso de inseticidas.

A Embrapa e Emater-PR aplicaram o MIP em 612 Unidades de Referências (URs) nas safras de 2013/14 a 2017/18 em lavouras de soja, e o resultado obtido foi a redução pela metade do uso de inseticidas e produtividade igual às que não utilizam a tecnologia, levando a uma redução média de custo de 2,3 sacas de soja/ha (Conte et al., 2019).

O aplicativo manejo.app desenvolvido pela UTFPR como resultado de um projeto de inovação tem sido usado para apoiar a coleta e análise de dados do MIP. O manejo.app garante maior eficiência por meio da automatização desse processo. Sem o uso do manejo.app, a análise dos dados podia durar até seis meses para ser finalizada, pois os dados eram coletados em uma planilha de papel e depois tabulados em planilhas eletrônicas sem a validação de dados gerando inconsistências negativas na análise dos dados (Silva, 2020).

Com o uso do manejo.app o técnico rural coloca os dados diretamente na aplicação permitindo a visualização e análise dos dados de forma instantânea. Dessa forma, a análise de dados pode ser realizada em qualquer etapa da safra e também permite o acompanhamento de um maior número de lavouras sem prejudicar a qualidade da coleta dos dados (Silva, 2020).

Contudo, existe a necessidade de melhorar a qualidade do manejo.app e garantir que novas funcionalidades não impactem de forma negativa no código já existente. Portanto, é fundamental criar testes automatizados para garantir mais qualidade e agilidade em futuras modificações na aplicação, preservando a integridade das regras de negócio.

Na execução do presente trabalho, foram desenvolvidos 365 testes unitários para a aplicação. Com o uso do JUnit, os testes cobrem a camada de entidades (`entity`), que consiste dos pacotes `base`, `mid`, `mip`, `pulverisation` e `survey`.

MATERIAL E MÉTODOS

O JUnit é um *framework* de código aberto para desenvolvimento de testes unitários em Java. O *framework* pode ser integrado a diversas IDEs para facilitar sua utilização e visualização dos resultados (Gulati e Sharma, 2017). Algumas das características do JUnit são: (i) facilidade de aprendizado e de escrita dos testes, (ii) fácil e rápida execução dos testes, (iii) testes isolados, evitando interferência entre eles, (iv) comparação de resultados reais com os resultados esperados em cada teste e (v) fácil visualização dos resultados (Beck, 2004).

Os seguintes passos foram necessários para realizar este projeto:

a) Entender o funcionamento do *framework* JUnit: o *framework* consiste de diversas instruções que, em conjunto, permitem o teste da aplicação.

Dessa forma, foi necessário conhecer essas instruções e seu funcionamento;

b) Analisar as classes presentes nos pacotes `base`, `mid`, `mip`, `pulverisation` e `survey`. Os pacotes consistem de diversas classes. Dessa forma, foi necessário analisar as classes e seus relacionamentos antes de desenvolver os testes;

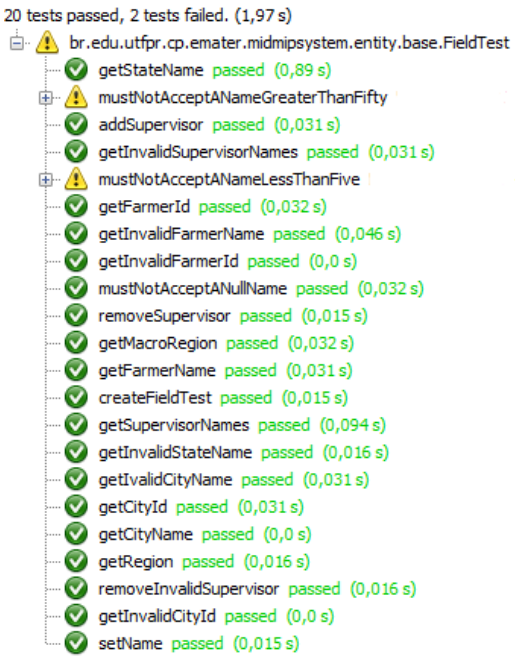
c) Desenvolver testes unitários. Cada teste avalia resultados esperados e reais para os métodos públicos de cada classe nos cinco pacotes analisados.

RESULTADOS E DISCUSSÃO

Foram realizados os testes das classes dos pacotes `base`, `mid`, `mip`, `pulverisation` e `survey`, pertencentes à camada `entity`. Essa camada representa as classes de domínio da aplicação. Como a tarefa de desenvolvimento e execução dos testes é repetitiva, esta seção se concentra em mostrar os testes mais relevantes para a classe `Field`, do pacote `base`.

Para o pacote `base`, foram criados 79 testes, dos quais 4 apresentaram problemas por conta de erros nas classes do pacote. A Figura 1 mostra o resultado de execução dos testes para a classe `Field`, a principal classe desse pacote. Para essa classe, foram realizados 22 testes. Os testes abrangem métodos públicos da classe. A Figura 1 mostra que a maioria dos testes não identificaram erros (ícone de confirmação, em verde), enquanto dois testes apresentaram problemas (ícone de alerta, em amarelo). Os dois testes que falharam foram de validação do atributo `name`.

Figura 1 – Testes da classe *Field*



Fonte: Captura de tela do Eclipse exibindo o resultado da execução dos testes unitários para a classe *Field* (2020).

A Figura 2 mostra o código dos dois testes que falharam. O objetivo dos testes é validar um nome informado, que deve ter entre cinco e cinquenta caracteres. Para o primeiro teste, que vai da linha 60 a 67, foi colocado um nome menor que cinco caracteres – como pode ser observado na linha 62. No segundo teste, que vai da linha 69 a 76, foi colocado um nome maior que cinquenta caracteres – conforme observado na linha 71. As linhas 63 e 72 realizam a validação do objeto Field. Por fim, as linhas 66 e 75 verificam se houve violação da regra, por meio da verificação de uma mensagem. Os testes falharam porque as validações no código da classe `Field` disparam uma mensagem diferente da mensagem esperada pelo teste. Enquanto a classe `Field` apresenta uma mensagem "O nome da região deve ter entre 5 e 50 caracteres", o teste da Figura 2 apresenta "O nome do campo deve ter entre 5 e 50 caracteres.

Figura 2 - Teste de validação do atributo *name*

```
60          @Test
61          public void mustNotAcceptANameLessThanFive() {
62                  field.setName("ab");
63                  Set<ConstraintViolation<Field>> violations = validator.validate(field);
64
65                  assertThat(violations.toString())
66                          .contains("O nome do campo deve ter entre 5 e 50 caracteres");
67          }
68
69          @Test
70          public void mustNotAcceptANameGreaterThanFifty() {
71                  field.setName("abababababab abababababab abababababab abababababab abababababab");
72                  Set<ConstraintViolation<Field>> violations = validator.validate(field);
73
74                  assertThat(violations.toString())
75                          .contains("O nome do campo deve ter entre 5 e 50 caracteres");
76          }
```

Fonte: Captura de tela do Eclipse exibindo o código-fonte da aplicação manejo.app (2020).

A Figura 3 mostra o teste que verifica a adição de um responsável técnico (`addSupervisor`). O método na classe `Field` deve criar uma lista com os supervisores, caso não exista, e também deve permitir adicionar mais de um supervisor. Para efeito de teste, na linha 88 é criado um supervisor. Em seguida, na linha 92 é usado o método `addSupervisor` para adicionar ele no objeto `Field`. Por fim, na linha 94 é verificado se o retorno do método `getSupervisorNames` é igual a uma lista contendo o nome do supervisor.

Figura 3 – Teste do método *addSupervisor*

```
86          @Test
87          public void addSupervisor() {
88                  Supervisor supervisor1 = Supervisor.builder().id(11)
89                          .email("larimaroli@emater.pr.gov.br").name("Lari Maroli")
90                          .region(createValidRegion()).build();
91
92                  field.addSupervisor(supervisor1);
93
94                  assertThat(field.getSupervisorNames()
95                          .toString()).isEqualTo("[Lari Maroli]");
96          }
```

Fonte: Captura de tela do Eclipse exibindo o código-fonte da aplicação manejo.app (2020).

A Figura 4 mostra dois testes para o método `removeSupervisor`, que deve excluir o supervisor escolhido e retornar verdadeiro. Caso o supervisor escolhido não esteja na lista, deve retornar falso. O primeiro teste (linhas 98 a 113) cria e adiciona os valores definidos para `supervisor1` e `supervisor2` no objeto

`Field`, conforme pode ser observado nas linhas 100-107. Depois, o teste executa o método que deve excluir o `supervisor2`, salvando o valor do retorno na variável `var`, na linha 109. Por fim, o teste verifica se a lista dos nomes dos supervisores contém apenas o nome definido para `supervisor1`, e se o valor de `var` é verdadeiro.

O segundo teste (linhas 115 a 124) cria o `supervisor1` na linha 117. Porém, o `supervisor1` não é adicionado na lista. A linha 121 executa o método para remover o `supervisor1` e o retorno é guardado na variável `var`. Por fim, na linha 123 o teste verifica o valor da variável `var` se é falso. Conforme a Figura 1, os dois testes passaram validando o método.

Figura 4 - Testes do método *removeSupervisor*.

```
98      @Test
99      public void removeSupervisor() {
100             Supervisor supervisor1 = Supervisor.builder().id(11)
101                     .email("larimaroli@emater.pr.gov.br").name("Lari Maroli")
102                     .region(createValidRegion()).build();
103             Supervisor supervisor2 = Supervisor.builder().id(11)
104                     .email("grcornelio@emater.pr.gov.br").name("Eliani Aparecida Marson")
105                     .region(createValidRegion()).build();
106             field.addSupervisor(supervisor1);
107             field.addSupervisor(supervisor2);
108
109             boolean var = field.removeSupervisor(supervisor2);
110
111             assertThat(field.getSupervisorNames().toString()).isEqualTo("[Lari Maroli]");
112             assertThat(var).isTrue();
113     }
114
115     @Test
116     public void removeInvalidSupervisor() {
117             Supervisor supervisor1 = Supervisor.builder().id(11)
118                     .email("larimaroli@emater.pr.gov.br").name("Lari Maroli")
119                     .region(createValidRegion()).build();
120
121             boolean var = field.removeSupervisor(supervisor1);
122
123             assertThat(var).isFalse();
124     }
```

Fonte: Captura de tela do Eclipse exibindo o código-fonte da aplicação manejo.app (2020).

CONCLUSÃO

A criação dos testes com a utilização do JUnit foi simples, porém trabalhosa. Como a utilização dos recursos do JUnit é simples, fica fácil estruturar os testes. A verificação do resultado esperado é intuitiva, facilitando a legibilidade e a escrita do teste. Outra facilidade é que a anotação `@Test` pode receber o parâmetro `expected` para verificar se uma exceção é lançada. A anotação `@Before` também é útil pois diminui a repetição de código.

Uma vez que alguns testes desenvolvidos neste projeto falharam, isso mostra que o código do manejo.app contém algumas falhas. Dessa forma, fica evidente que esse projeto contribuiu para mostrar as falhas que precisam ser corrigidas. Adicionalmente, os testes criados neste projeto contribuem para a adição de novas funcionalidades de forma segura, pois é possível verificar se funcionalidades existentes foram afetadas.

Como participante de um projeto de inovação, consegui aprender na prática vários conceitos e tecnologias, como: (i) programação orientada a objetos

avançada, (ii) testes unitários, como teste de caixa branca e análise de valor limite, (iii) Java Server Faces (JSF), (iv) Spring Boot e (v) JUnit. Esses conceitos e tecnologias foram necessários para compreensão e desenvolvimento dos testes. Além disso, também tive a oportunidade de ter obtido conhecimentos práticos de como é feito o manejo integrado de pragas e suas vantagens.

AGRADECIMENTOS

O presente trabalho foi realizado com o apoio do Conselho Nacional de Desenvolvimento Científico e Tecnológico CNPq – Brasil. Como bolsista de iniciação em desenvolvimento tecnológico e inovação, agradeço ao CNPq pela bolsa de estudos.

REFERÊNCIAS

BECK, Kent. **JUnit Pocket Guide**. Sebastopol: O'Reilly Media, 2004.

CONTE, O.; OLIVEIRA, F. T. de; HARGER, N.; CORRÊA-FERREIRA, B. S.; ROGGIA, S.; PRANDO, A. M.; POSSAMAI, E. J.; REIS, E. A.; MARX, E. F. **Resultados do manejo integrado de pragas da soja na safra 2018/19 no Paraná.** Londrina: EMBRAPA/EMATER, 2019.

GULATI, S.; SHARMA, R. **Java Unit Testing with JUnit 5**: test driven development with junit 5. Nova Iorque: Apress, 2017.

SILVA, G. C. **Desenvolvimento de aplicativos para coleta e análise de dados da tecnologia de manejo integrado de pragas da cultura da soja**. Cornélio Procópio, 2020.