

**UNIVERSIDADE FEDERAL DE SÃO CARLOS**  
**DEPARTAMENTO DE COMPUTAÇÃO**

**ENGENHARIA DE SOFTWARE 2**

**TRABALHO PRÁTICO - JABREF**

**Prof. Auri Vincenzi**

Bruno Guerra Dias Pereira- 489360

Jéssica Caroline Dias Nascimento - 489336

João Vieira da Silva Neto - 610054

Romão Matheus Martines de Jesus - 595071

**SÃO CARLOS**

**2017/1**

## 1 INÍCIO DA ANÁLISE

Primeiramente fizemos a coberturas do código utilizando a ferramenta sonarqube (Figura 1 e Figura 2) para podermos ter noção da quantidade de linhas testadas e qual era a cobertura atual de testes do nosso programa.

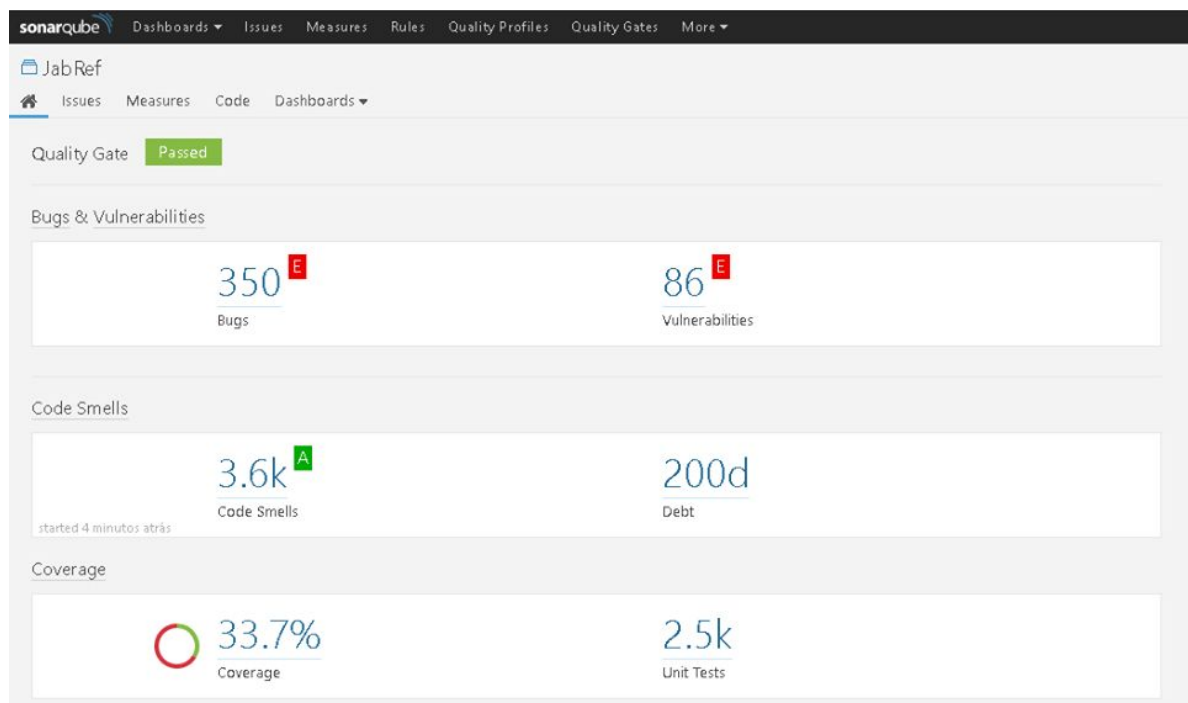


Figura 1: Parte 1 tela do sonarqube.

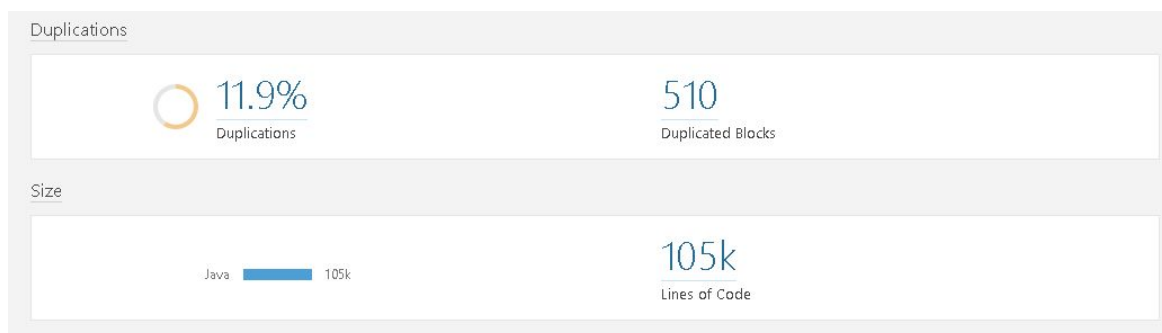


Figura 2: Parte 2 tela do sonarqube.

## 2 MANUTENÇÃO PERFECTIVA DA INSERÇÃO DE NOVOS ITENS

No projeto original, não havia nenhum tipo de verificação da validade do ano, na inserção de novos itens e nem da chave que serviria como id de cada novo item inserido.

### 2.1 Validação de chave

O único comando da chave no arquivo *BibEntry.java* no método *setCiteKey()* era o de preenchimento independente do conteúdo da chave (Figura 3).

```
public void setCiteKey(String newCiteKey) {  
    setField(KEY_FIELD, newCiteKey);  
}
```

Figura 3: método setCiteKey()

Isso permitia que a chave de cada item pudesse ser um número, uma composição de caracteres especiais, e ainda poderia ser vazia, o que complicaria o padrão de chaves dos itens.

Segundo a especificação do trabalho, as novas verificações implementadas neste mesmo método fariam a chave ter de possuir ao menos dois caracteres e começar com alguma letra maiúscula ou minúscula, caso contrário a chave era dada como inválida (Figura 4).

```
public void setCiteKey(String newCiteKey) {  
    // Adicionando a verificacao de entrada de caracteres: deve ser no minimo 2 e comecar com uma letra  
    try {  
        if (!(newCiteKey.length() >= 2) && Character.isLetter(newCiteKey.charAt(0))) {  
            throw new ParseException("", 0);  
        }  
        setField(KEY_FIELD, newCiteKey);  
    } catch (ParseException e) {  
        throw new IllegalArgumentException("Entrada Invalida! Deve-se possuir ao menos dois caracteres e deve-se" +  
            "comecar com caracteres textuais");  
    }  
}
```

Figura 4: novas verificações do CiteKey()

A partir daqui, ao inserir uma chave inválida, ou seja, que não cumpria os requisitos, um erro era lançado ao sistema através da diretiva *IllegalArgumentException()*.

## 2.2 Validação de ano

No ano, na configuração original sequer havia verificação, sendo assim o campo poderia também ser preenchido com qualquer item, inclusive podendo estar vazio.

Para a verificação do código, adicionamos dentro no método *setField()*, no mesmo arquivo, uma verificação através de try/catch para validar a entrada do ano (Figura 5).

```
if ("year".equals(name)) {  
    try {  
        if (!value.matches(regex: "^[0-9]+")) {  
            throw new ParseException("", 0);  
        }  
  
        SimpleDateFormat date_format = new java.text.SimpleDateFormat( pattern: "yyyy");  
        date_format.setLenient(false);  
        date_format.parse(value);  
    } catch (ParseException e) {  
        throw new IllegalArgumentException("Ano inválido!");  
    }  
}
```

Figura 5: adição da validação de ano.

Em caso de ano inválido, ou seja, sem ser da forma YYYY o campo de preenchimento fica em destaque (Figura 6), enviando uma exceção ao sistema.

The screenshot shows the BibTeX Quality Tools application interface. At the top, there is a menu bar with options like File, Edit, Search, Groups, View, BibTeX, Quality, Tools, Options, and Help. Below the menu is a toolbar with various icons for file operations and search. A search bar is located below the toolbar. The main area displays a table of entries with columns: #, entrytype, author/editor, title, year, journal/booktitle, bibtexkey, and ranking. Two entries are visible: an Article by Langley from 2017, and a Book by Rowling from 201002. Below the table, there is a detailed view of the selected entry, 'Harry Potter e a Pedra Filosofal'. The fields shown are Title, Publisher (Rocco), Year, Author (J. K. Rowling), Editor (Rocco), and Bibtexkey (fs). The 'Year' field is highlighted with a red background, indicating it is invalid. At the bottom, a status message reads: 'Status: BibTeXkey is unique.'

Figura 6: entrada inválida de ano

### 3 TESTES DA FUNCIONALIDADE DA INSERÇÃO

Para verificar o comportamento das verificações adicionadas (foram 26 novas linhas de código no arquivo *BibtexEntry.java*) criamos 18 novos casos de teste, totalizando mais 63 linhas de código criadas para estes, nos arquivos *BibEntryTest.java* e *BibEntryTests.java*.

Utilizando a técnica de classes de equivalência, tentou-se criar o menor número possível de casos de teste para a maior cobertura (Figura 7 e Figura 8).

```
51 // Teste de entrada invalida para texto com dois ou mais caracteres
52 @Test (expected = IllegalArgumentException.class)
53 public void setInvalidYearInvalidFieldText() { entry.setField( name: "year", value: "teste"); }
54
55 // Teste de entrada invalida para texto com numeros mas comecando com letras
56 @Test (expected = IllegalArgumentException.class)
57 public void setInvalidYearInvalidFieldTextNumber() { entry.setField( name: "year", value: "teste2017"); }
58
59 // Teste de entrada invalida para textos comecados com numeros
60 @Test (expected = IllegalArgumentException.class)
61 public void setInvalidYearInvalidFieldNumberText() { entry.setField( name: "year", value: "2017teste"); }
62
63 // Teste de entrada invalida para textos com caracteres especiais no inicio
64 @Test (expected = IllegalArgumentException.class)
65 public void setInvalidYearInvalidFieldNumberWithSpecialCharacter() { entry.setField( name: "year", value: "_2017"); }
66
67 // Teste de entrada invalida para textos com menos de dois caracteres
68 @Test (expected = IllegalArgumentException.class)
69 public void setInvalidYearInvalidFieldTextBelowTwoCharacters() { entry.setField( name: "year", value: "t"); }
70
71 // Teste de entrada invalida para textos com dois caracteres especiais
72 @Test
73 public void setInvalidYearInvalidFieldSpecialCharacters() { entry.setField( name: "year", value: "7"); }
74
75 // Teste de entrada invalida para textos com um digito apenas
76 @Test (expected = IllegalArgumentException.class)
77 public void setInvalidYearValidFieldText() { entry.setField( name: "year", value: "teste"); }
```

Figura 7: alguns testes de validação de ano

```
95 @Test
96 public void setValidBibtexKeyTextNumber() {
97     entry.setField( name: "bibtexkey", value: "ES22017");
98     Map<String, String> str2 = entry.getFieldMap();
99     assertEquals( expected: "ES22017", str2.get("bibtexkey"));
100 }
101
102 // Entrada valida de chave com mais de um caractere e apenas letras
103 @Test
104 public void setValidBibtexKeyOnlyText() {
105     entry.setField( name: "bibtexkey", value: "Aurj");
106     Map<String, String> str2 = entry.getFieldMap();
107     assertEquals( expected: "Aurj", str2.get("bibtexkey"));
108 }
109
110 // Entrada valida com caracteres e caracteres especiais
111 @Test
112 public void setValidBibtexKeySpecialCharacter() {
113     entry.setField( name: "bibtexkey", value: "ES22017!");
114     Map<String, String> str2 = entry.getFieldMap();
115     assertEquals( expected: "ES22017!", str2.get("bibtexkey"));
116 }
117
118 // Entrada valida com exatamente dois caracteres
119 @Test
120 public void setValidBibtexKeyTwoCharacters() {
121     entry.setField( name: "bibtexkey", value: "ES");
122     Map<String, String> str2 = entry.getFieldMap();
123     assertEquals( expected: "ES", str2.get("bibtexkey"));
124 }
```

Figura 8: alguns testes de validação da Bibtexkey

### 3.1 Relatório de cobertura de testes

Após realizar o teste de cobertura com o JaCoCo, chegamos aos resultados da Figura 9.

BibEntryTest	100% (1/ 1)	100% (21/ 21)	95.5% (42/ 44)
BibEntryTests	100% (1/ 1)	100% (58/ 58)	93.3% (208/ 223)

Figura 9: resultados após manutenção da inserção

Antes da manutenção, o teste de cobertura com o JaCoCo apresentava os resultados da Figura 10.

BibEntryTest	100% (1/ 1)	100% (7/ 7)	87.5% (14/ 16)
BibEntryTests	100% (1/ 1)	100% (54/ 54)	95.7% (198/ 207)

Figura 10: resultados antes da manutenção da inserção

## 4 MANUTENÇÃO PERFECTIVA DA INSERÇÃO DE CSV FILES

Para adicionar o comportamento da aceitação de arquivos CSV, foi criada a classe CSVImporter, que contém 223 linhas, realizando todos os requisitos que sua classe mãe Importer requer. O código analisa arquivos csv como input e verifica se os mesmos são válidos ou não para o JabRef. Foi necessária a modificação da classe FileExtensions.java (Figura 11) para a adição da extensão .csv.

```
@Override
public boolean isRecognizedFormat(BufferedReader reader) throws IOException {

    String str;
    while ((str = reader.readLine()) != null) {
        String[] fields = str.split(";", (?=([^\"]*\"[^\"]*\")*\"[^\"]*$)", -1);
        if ((fields.length % 2) == 0 || !RECOGNIZED.matcher(str).find()) {
            return false;
        }
    }
    return true;
}
```

Figura 11: método que verifica validade de csv utilizando o poder de expressões regulares da classe Pattern



Para testar o CSVImporter, foi criada a classe CSVImpoterTest (Figura 12 e Figura 13), que está contida no conjunto de testes das funcionalidades do package importer. CSVImpoterTest contém 101 linhas de código. Nosso CSVImporterTest realiza testes para verificar como o sistema responde tanto quando receber arquivos .csv corrompidos ou arquivos .csv no formato correto.

```
74 @Test
75 public void testCsvFields() throws IOException, URISyntaxException {
76     Path file = Paths.get(CSVImporterTest.class.getResource("csvFileWorking.csv").toURI());
77     List<BibEntry> bibEntries = importer.importDatabase(file, StandardCharsets.UTF_8).getDatabase().getEntries();
78
79     for (BibEntry entry : bibEntries) {
80
81         if (entry.getCiteKey().equals("small")) {
82             assertEquals(Optional.of("Guerra, B. D."), entry.getField( name: "author"));
83             assertEquals(Optional.of("ES2 é Amor"), entry.getField( name: "title"));
84             assertEquals(Optional.of("O Journal do Auri"), entry.getField( name: "journal"));
85             assertEquals(Optional.of("2017"), entry.getField( name: "year"));
86             assertEquals(Optional.of("-1"), entry.getField( name: "volume"));
87             assertEquals(Optional.of("to appear"), entry.getField( name: "note"));
88         } else if (entry.getCiteKey().equals("big")) {
89             assertEquals(Optional.of("Marx, Karl"), entry.getField( name: "author"));
90             assertEquals(Optional.of("A big communist paper"), entry.getField( name: "title"));
91             assertEquals(Optional.of("JOURNAL OF THE COMMUNISM WILL WIN"), entry.getField( name: "journal"));
92             assertEquals(Optional.of("1994"), entry.getField( name: "year"));
93             assertEquals(Optional.of("MCMXCVII"), entry.getField( name: "volume"));
94         }
95     }
96 }
97
```

**Figura 12: método da classe CSVImporterTest que testa a validação dos campos dos arquivos csv que foram importados pelo JabRef**

```
@Test
public void testIsRecognizedFormat() throws IOException, URISyntaxException {
    Path file = Paths.get(CSVImporterTest.class.getResource("csvFileWorking.csv").toURI());
    assertTrue(importer.isRecognizedFormat(file, StandardCharsets.UTF_8));
}

@Test
public void testIsRecognizedFormatReject() throws IOException, URISyntaxException {
    Path file = Paths.get(CSVImporterTest.class.getResource("csvFileNotWorking.csv").toURI());
    assertFalse(importer.isRecognizedFormat(file, StandardCharsets.UTF_8));
}
```

**Figura 13: método da classe CSVImporterTest que testa a validação do formato dos arquivos csv importado pelo JabRef**

#### 4.1 Testes para inserção de CSV

Nossa classe CSVImporterTest resultou em um desempenho de coverage bem satisfatório, como o relatório JaCoCo da Figura 14 pode mostrar.

## Coverage Summary for Class: CSVImporter (net.sf.jabref.logic.importer.fileformat)

Class	Class, %	Method, %	Line, %
CSVImporter	100% (1/ 1)	100% (8/ 8)	66.1% (78/ 118)

**Figura 14: relatório dos testes da inserção de CSV**

## 5 MANUTENÇÃO PERFECTIVA PARA ENTRADAS DUPLICADAS

Para adicionar o comportamento de oferecer a opção a criação de um novo database quando houverem entradas duplicadas no JabRef, nenhuma classe foi criada (além de classes para testes, é claro). A classe modificada para realizar a manutenção foi a ImportInspectionDialog (Figura 15).

```

if (Globals.prefs.getBoolean(JabRefPreferences.WARN_ABOUT_DUPLICATES_IN_INSPECTION)) {
    PrintWriter newFile = null;
    try {
        newFile = new PrintWriter("bin/tmp/temporaryNewDatabase", "UTF-8");
        for (BibEntry entry : entries) {
            newFile.println(entry);
            newFile.print("\n");
        }
        newFile.close();
    } catch (FileNotFoundException | UnsupportedEncodingException e) {
    }

    for (BibEntry entry : entries) {
        // Only check entries that are to be imported. Keep status
        // is indicated
        // through the search hit status of the entry:
        if (!entry.isSearchHit()) {
            continue;
        }

        // Check if the entry is a suspected, unresolved, duplicate.
        // This status
        // is indicated by the entry's group hit status:
        if (entry.isGroupHit()) {
            CheckBoxMessage cbm = new CheckBoxMessage(
                Localization.lang( key: "There are possible duplicates (marked with an icon) that haven't been resolved. " +
                    "Do you want to proceed (yes) or add them to a new database (no)?"),
                Localization.lang( key: "Disable this confirmation dialog"), defaultValue: false);
            int answer = JOptionPane.showConfirmDialog( component: ImportInspectionDialog.this, cbm,
                Localization.lang( key: "Duplicates found"), JOptionPane.YES_NO_OPTION);
            if (cbm.isSelected()) {
                Globals.prefs.putBoolean(JabRefPreferences.WARN_ABOUT_DUPLICATES_IN_INSPECTION, false);
            }
            if (answer == JOptionPane.NO_OPTION) {
                // neste ponto chamaremos novo database
                ImportMenuItem imi = new ImportMenuItem(frame, openInNew: true, importer: null);
                imi.automatedImport(Collections.singletonList("bin/tmp/temporaryNewDatabase"));
                BasePanel newPanel = (BasePanel) frame.getTabbedPane().getSelectedComponent();
                newPanel.getBibDatabaseContext().setDatabaseFile(null);
                newPanel.markBaseChanged();

                dispose();
                return;
            }
        }
    }
}

```

**Figura 15: modificação realizada na ImportInspectionDialog para adicionar requisito da manutenção**



Criamos um novo database que será armazenado em uma pasta temporária, na qual o usuário poderá decidir o que quiser fazer depois. Feita a manutenção, realizamos em seguida os testes necessários para poder automatizar sua validação. Para tal, foi criada a classe `DuplicateImportTest` (Figura 16), que possui 78 linhas, na qual foi realizada uma herança da classe `AbstractUITest`, que possui métodos que foram essenciais para as realizações de testes. O teste utiliza da função `robot()`, basicamente um robô que irá performar as ações previamente estabelecidas.

```
25 public class DuplicateImportTest extends AbstractUITest {
26
27     @Override
28     protected void setUp() {
29         awtExceptionHandler = new AWTExceptionHandler();
30         awtExceptionHandler.installExceptionDetectionInEDT();
31         application(JabRefMain.class).start();
32
33         robot().waitForIdle();
34         robot().settings().timeoutToFindSubMenu( ms: 1_000);
35         robot().settings().delayBetweenEvents(SPEED_NORMAL);
36
37         mainFrame = findFrame(JabRefFrame.class).withTimeout(10_000).using(robot());
38         robot().waitForIdle();
39     }
40
41
42     private void importBibIntoCurrentDatabase(String path) {
43         mainFrame.menuItemWithPath("File", "Import into current database").click();
44         JFileChooserFixture openFileDialog = mainFrame.fileChooser();
45         robot().settings().delayBetweenEvents( ms: 1);
46         openFileDialog.fileNameTextBox().enterText(path);
47         openFileDialog.approve();
48         Pause.pause( ms: 1_000);
49     }
```

**Figura 16: definition da classe `DuplicateImportTest`, contendo seu `SetUp` e o método `importBibIntoCurrentDatabase`, que basicamente é um método ‘gêmeo’ de `importBibIntoNewDatabase`**

Foi pedido como requisito que fosse dada prioridade para testes com arquivos `.bib` ou `.csv`, e para aproveitar e utilizar da manutenção realizada previamente, a de importar arquivos `.csv`, foi decidido pelo grupo que fosse utilizado arquivos `.csv` para testes (Figura 17).

```

@Test
public void testDuplicateEntries() throws IOException {

    String path = new File(this.getClass().getClassLoader().
        getResource("net/sf/jabref/logic/importer/fileformat/csvFileWorking.csv").getFile()).getAbsolutePath();
    importBibIntoNewDatabase(path);
    importBibIntoCurrentDatabase(path);
    DialogFixture importInspectionDialog = findDialog(ImportInspectionDialog.class).withTimeout(10000).using(robot());

    importInspectionDialog.button(new GenericTypeMatcher<JButton>(JButton.class) {
        @Override
        protected boolean isMatching(@Nonnull JButton jButton) { return "OK".equals(jButton.getText()); }
    }).click();
    robot().settings().delayBetweenEvents(1000);
    importInspectionDialog.button(new GenericTypeMatcher<JButton>(JButton.class) {
        @Override
        protected boolean isMatching(@Nonnull JButton jButton) { return "No".equals(jButton.getText()); }
    }).click();

    exitJabRef();
}

```

**Figura 17: método teste que irá utilizar o robot() para realizar os passos necessários para a validação da implementação realizada**

### 5.1 Testes para entradas duplicadas

Nossa classe DuplicateImportTest resultou em um desempenho de coverage bem satisfatório para o trecho de código criado e/ou modificado (as linhas em verde são as que estão no escopo de teste, enquanto as em vermelho não), como o relatório JaCoCo da Figura 18 e Figura 19 podem mostrar.

```

620         @Override
621         public void actionPerformed(ActionEvent event) {
622
623             // First check if we are supposed to warn about duplicates. If so,
624             // see if there
625             // are unresolved duplicates, and warn if yes.
626             if (Globals.prefs.getBoolean(JabRefPreferences.WARN_ABOUT_DUPLICATES_IN_INSPECTION)) {
627
628                 PrintWriter newFile = null;
629                 try {
630                     newFile = new PrintWriter("bin/tmp/temporaryNewDatabase", "UTF-8");
631                     for (BibEntry entry : entries) {
632                         newFile.println(entry);
633                         newFile.print("\n");
634                     }
635                     newFile.close();
636                 } catch (FileNotFoundException | UnsupportedEncodingException e) {
637                 }
638
639                 for (BibEntry entry : entries) {
640
641                     // Only check entries that are to be imported. Keep status
642                     // is indicated
643                     // through the search hit status of the entry:
644                     if (!entry.isSearchHit()) {
645                         continue;
646                     }

```

**Figura 18: primeira parte do relatório de testes para entradas duplicadas**

```

647
648 // Check if the entry is a suspected, unresolved, duplicate.
649 // This status
650 // is indicated by the entry's group hit status:
651 if (entry.isGroupHit()) {
652     CheckBoxMessage cbm = new CheckBoxMessage(
653         Localization.lang("There are possible duplicates (marked with an icon) that haven't b
654         Localization.lang("Disable this confirmation dialog"), false);
655     int answer = JOptionPane.showConfirmDialog(ImportInspectionDialog.this, cbm,
656         Localization.lang("Duplicates found"), JOptionPane.YES_NO_OPTION);
657     if (cbm.isSelected()) {
658         Globals.prefs.putBoolean(JabRefPreferences.WARN ABOUT DUPLICATES IN INSPECTION, false);
659     }
660     if (answer == JOptionPane.NO_OPTION) {
661         // neste ponto chamaremos novo database
662         ImportMenuItem imi = new ImportMenuItem(frame, true, null);
663         imi.automatedImport(Collections.singletonList("bin/tmp/temporaryNewDatabase"));
664         BasePanel newPanel = (BasePanel) frame.getTabbedPane().getSelectedComponent();
665         newPanel.getBibDatabaseContext().setDatabaseFile(null);
666         newPanel.markBaseChanged();
667     }
668     dispose();
669     return;
670 }
671 break;
672 }
673 }
674 }

```

Figura 19: segunda parte do relatório de testes para entradas duplicadas

## 6 MANUTENÇÃO PERFECTIVA NOME DO AUTOR

No projeto original, ao inserir os dados para a inserção de um *BibTexEntry*, percebe-se que a entrada de dados é do tipo string e aceita qualquer caracter (Figura 20), permitindo a inserção de números e caracteres especiais.

O objetivo da manutenção perfectiva é permitir que o usuário insira somente nomes válidos de acordo com as normas de referências. A manutenção foi feita no método *setField()* do arquivo *BibEntry*<sup>1</sup> e foram adicionadas 8 linhas de código, contando com importação da biblioteca *javax.swing.\** e comentário.

<sup>1</sup> src/main/java/net.sf.jabref/model/entry





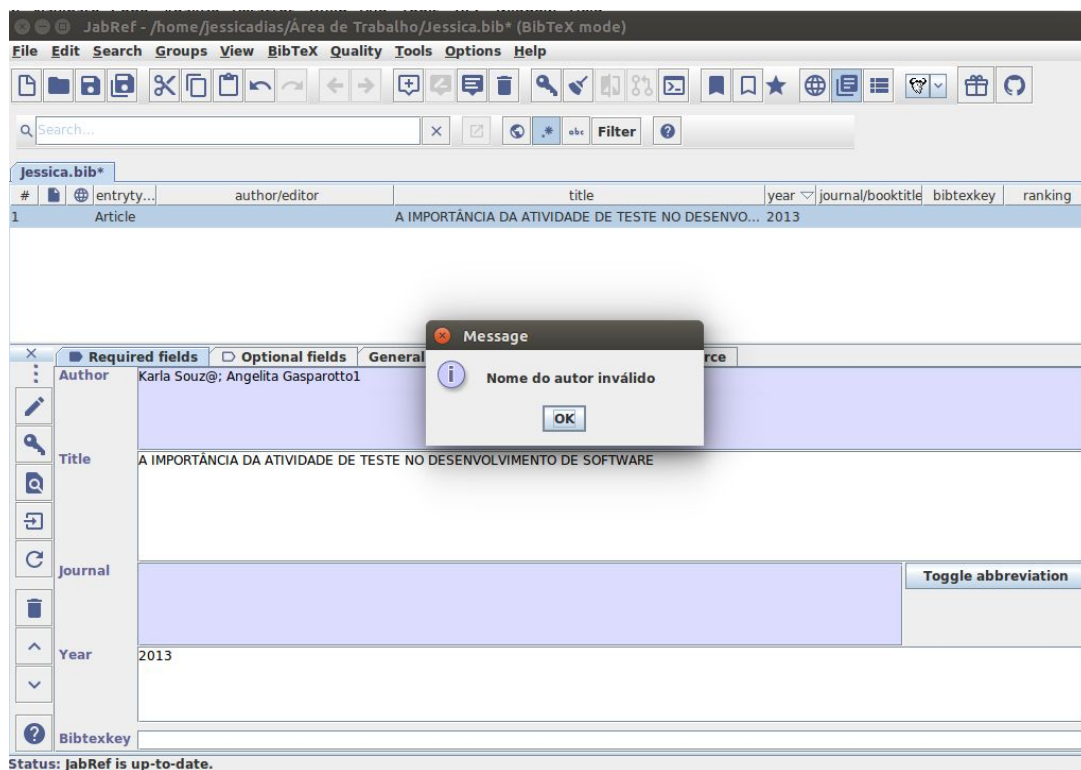


Figura 22: erro ao inserir nome inválido

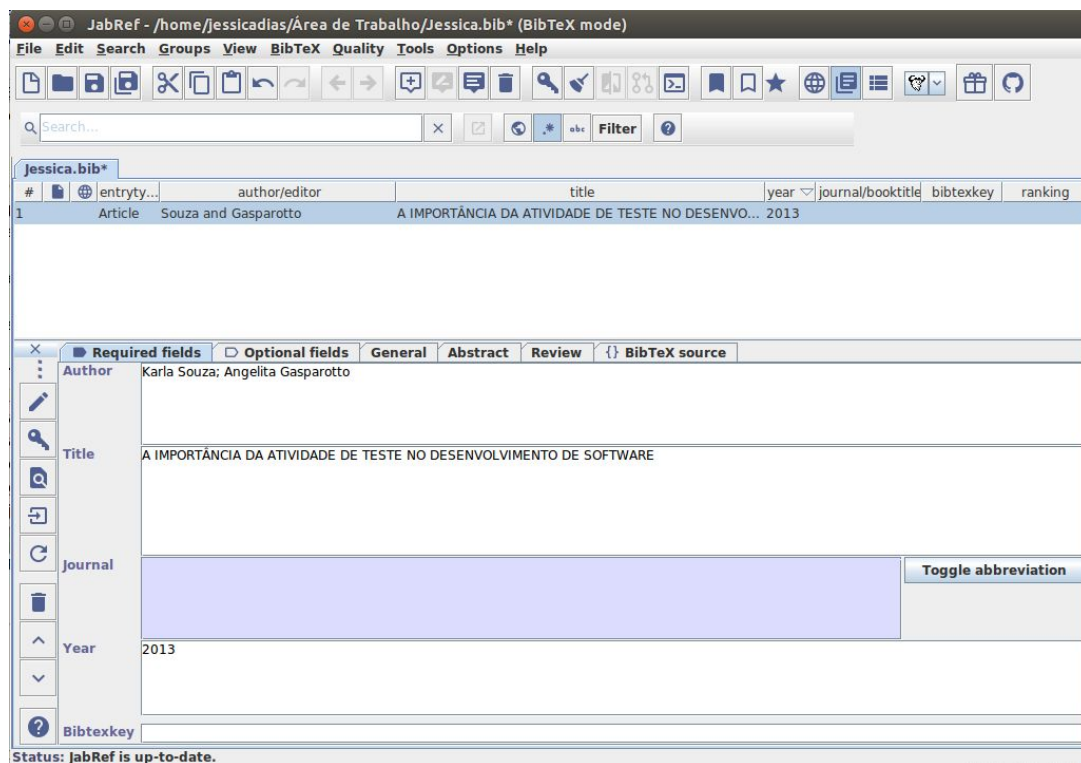
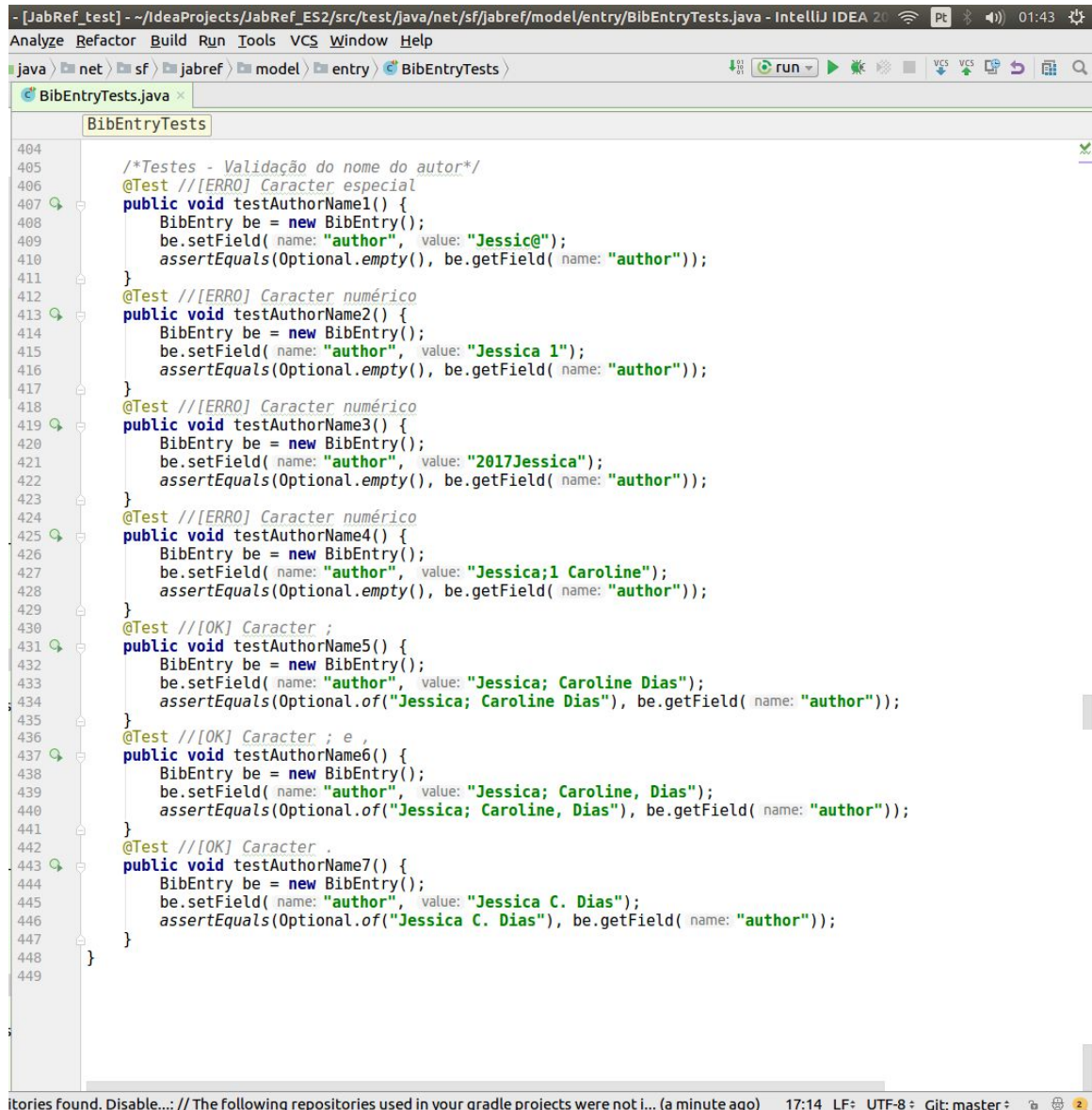


Figura 23: nome do autor correto

## 6.1 TESTES DO NOME DO AUTOR

O teste para a manutenção da funcionalidade nome do autor foi feito no arquivo *BibEntryTests*<sup>2</sup> e foram adicionadas 42 linhas de código (incluindo comentário) em 7 testes desenvolvidos. Os testes consistem em verificações de caracteres especiais e números (Figura 24).



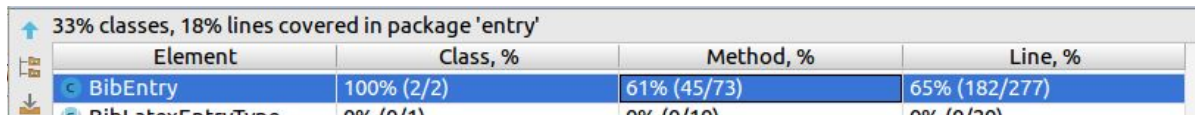
```
404      /*Testes - Validação do nome do autor*/
405      @Test //[[ERRO] Caracter especial
406      public void testAuthorName1() {
407          BibEntry be = new BibEntry();
408          be.setField( name: "author", value: "Jessic@");
409          assertEquals(Optional.empty(), be.getField( name: "author"));
410      }
411
412      @Test //[[ERRO] Caracter numérico
413      public void testAuthorName2() {
414          BibEntry be = new BibEntry();
415          be.setField( name: "author", value: "Jessica 1");
416          assertEquals(Optional.empty(), be.getField( name: "author"));
417      }
418
419      @Test //[[ERRO] Caracter numérico
420      public void testAuthorName3() {
421          BibEntry be = new BibEntry();
422          be.setField( name: "author", value: "2017Jessica");
423          assertEquals(Optional.empty(), be.getField( name: "author"));
424      }
425
426      @Test //[[ERRO] Caracter numérico
427      public void testAuthorName4() {
428          BibEntry be = new BibEntry();
429          be.setField( name: "author", value: "Jessica;1 Caroline");
430          assertEquals(Optional.empty(), be.getField( name: "author"));
431      }
432
433      @Test //[[OK] Caracter ;
434      public void testAuthorName5() {
435          BibEntry be = new BibEntry();
436          be.setField( name: "author", value: "Jessica; Caroline Dias");
437          assertEquals(Optional.of("Jessica; Caroline Dias"), be.getField( name: "author"));
438      }
439
440      @Test //[[OK] Caracter ; e
441      public void testAuthorName6() {
442          BibEntry be = new BibEntry();
443          be.setField( name: "author", value: "Jessica; Caroline, Dias");
444          assertEquals(Optional.of("Jessica; Caroline, Dias"), be.getField( name: "author"));
445      }
446
447      @Test //[[OK] Caracter .
448      public void testAuthorName7() {
449          BibEntry be = new BibEntry();
450          be.setField( name: "author", value: "Jessica C. Dias");
451          assertEquals(Optional.of("Jessica C. Dias"), be.getField( name: "author"));
452      }
453      }
```

Figura 24: código testes manutenção nome do autor

<sup>2</sup> src/test/java/net.sf.jabref/model/entry

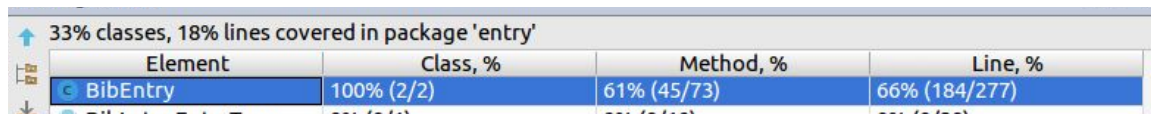


Ao executar o projeto com cobertura de código por meio do JaCoCo, pode-se observar que antes da inserção dos testes (Figura 25) a cobertura de código era de 61% dos métodos e 65% das linhas. Após a inserção dos testes do nome do autor (Figura 26), a cobertura de código foi modificada para 61% dos métodos e 66% das linhas de código.



33% classes, 18% lines covered in package 'entry'			
Element	Class, %	Method, %	Line, %
BibEntry	100% (2/2)	61% (45/73)	65% (182/277)

**Figura 25: cobertura do código antes dos testes da manutenção nome do autor**

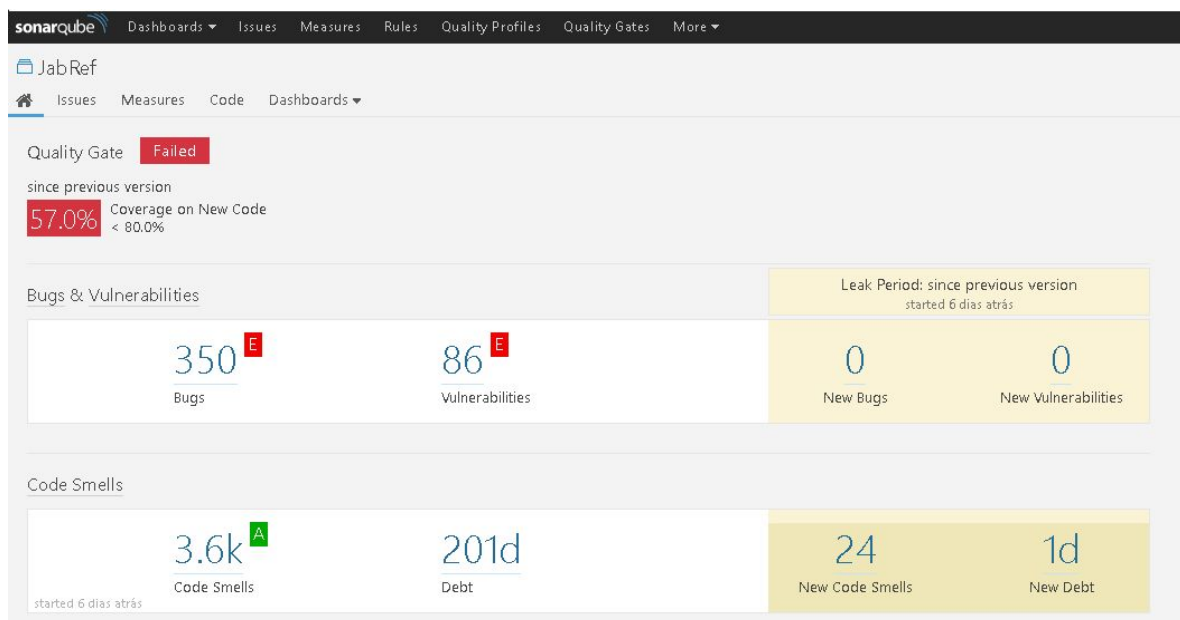


33% classes, 18% lines covered in package 'entry'			
Element	Class, %	Method, %	Line, %
BibEntry	100% (2/2)	61% (45/73)	66% (184/277)

**Figura 26: cobertura do código depois dos testes da manutenção nome do autor**

## 7 VISÃO FINAL DO PROJETO

Com todos os códigos adicionados e analisados não obtivemos uma grande cobertura do código, pois o programa é muito extenso: possui muitas linhas de código e classes a serem analisadas, com vários ramos e nós. Dessa maneira, a parte analisada e implementada do código neste trabalho não consegue aumentar significativamente a cobertura de todo o programa (Figura 27).



**Figura 27: Parte 1 tela do sonarqube.**

Podemos observar (Figura 28) um aumento de 0,1% de cobertura de código devido à manutenção de importação de CSV files e o aumento das linhas de código.



**Figura 28: Parte 2 tela do sonarqube.**

## 8 CONSIDERAÇÕES FINAIS

Percebemos analisando os relatórios de cobertura que olhando o código como um todo, não há grandes mudanças de cobertura, pois fazemos pequenas alterações em módulos muito específicos. Além disso, as linhas de código adicionadas possuem casos de teste que são criados pensando em cobrir exatamente estas linhas adicionadas, sendo assim, a cobertura dos módulos anteriores continua a mesma, pois temos uma configuração com mais linhas de códigos, mais métodos, porém os módulos que não eram cobertos anteriormente continuam sem cobertura.

Uma possível solução para isso seria pensar em casos de teste que cobrissem as linhas de código não contempladas por este trabalho, o que demandaria um estudo das funções e provavelmente acarretaria em novas manutenções perfectivas no código, reforçando a ideia de que um sistema nunca será perfeito.