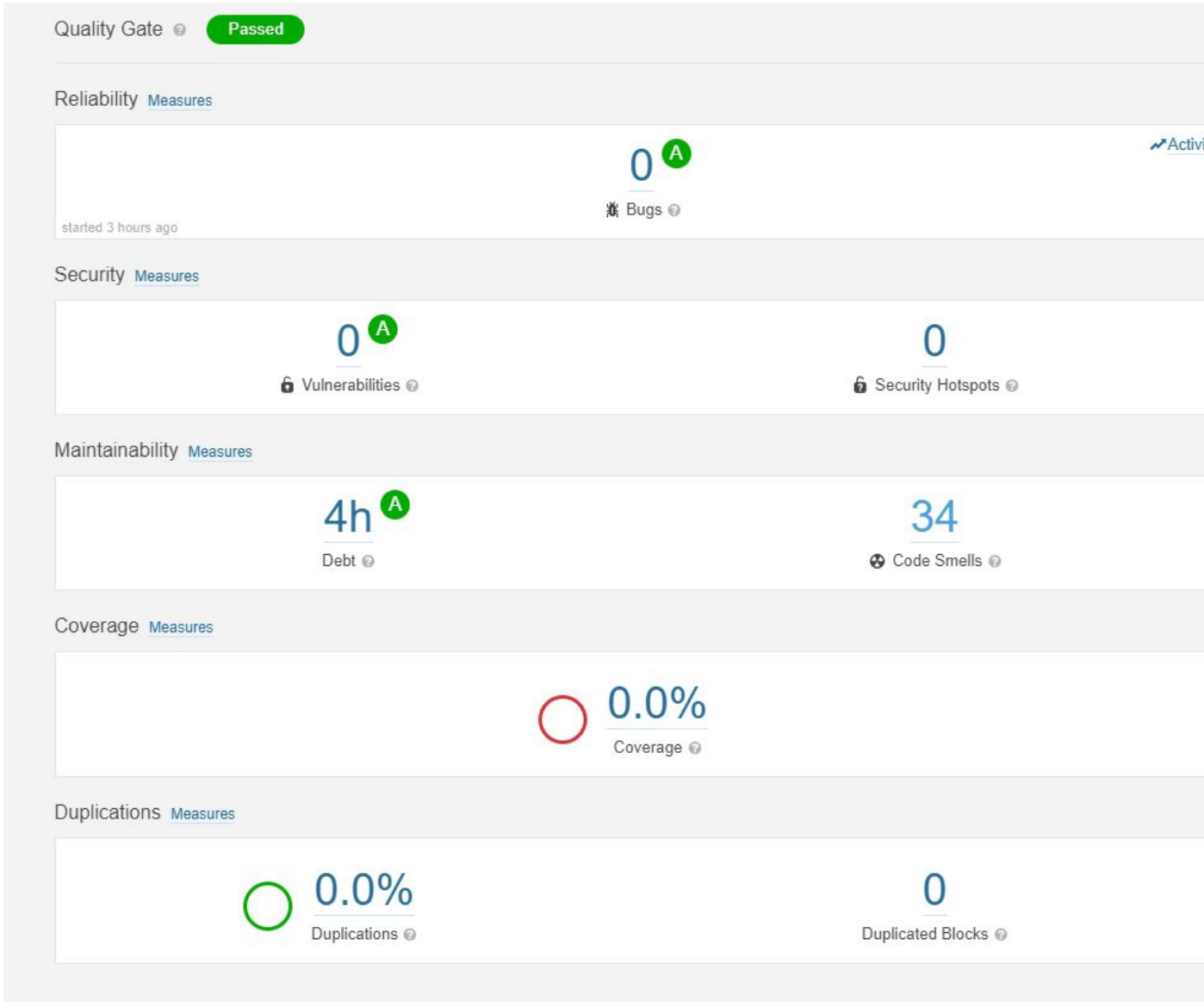


Código analisado com a API Sonarube em 29/10/2019

Análise geral do código



Code Smells presentes no código

Erros sublinhados de vermelho

A parte de importações de um arquivo deve ser manipulada pelo IDE, não manualmente.

Importações não utilizadas e inúteis não devem ocorrer, se for o caso. Deixá-los reduz a legibilidade do código, pois a presença deles pode ser confusa.

```
1  ... package com.ufrpe.bsi.soresenha.eventos.gui;
2
3  import android.content.Context;
4  import android.content.Intent;
5  import android.os.Build;
6
7  import android.support.annotation.NonNull;
8  import android.support.v7.widget.RecyclerView;
9  import android.view.ContextMenu;
10
11 import android.view.Gravity;
12
13 import android.view.LayoutInflater;
14 import android.view.MenuInflater;
15
16 import android.view.MenuItem;
17 import android.view.View;
18 import android.view.ViewGroup;
19 import android.widget.ImageButton;
```

```

16     import android.widget.PopupMenu;
17     import android.widget.TextView;
18     import android.widget.Toast;

```

O requisito para um default é a programação defensiva. O método deve tomar as medidas apropriadas ou conter um comentário adequado sobre o motivo pelo qual nenhuma ação foi tomada.

```

private void popupActions(PopupMenu popup, final int position) {
    popup.setOnMenuItemClickListener(new PopupMenu.OnMenuItemClickListener() {
        @Override
        public boolean onMenuItemClick(MenuItem menuItem) {
            switch (menuItem.getItemId()) {

```

As instruções switch são úteis quando existem muitos casos diferentes, dependendo do valor da mesma expressão.

Porém, apenas para um ou dois casos, o código ficará mais legível com as instruções if.

```

        case R.id.editEvent:
            moveToEdit(position);
            break;
        case R.id.deleteEvent:
            deleteEvent(position);
            break;
    }
    return false;
}
});
}

```

Classes internas anônimas deve ser substituída por lambdas para aumentar muito a legibilidade do código-fonte.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_splash_screen);

    getSupportActionBar().hide();

    getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN, WindowManager.LayoutParams.

    new Handler().postDelayed(new Runnable() {
```

Strings literais duplicadas tornam o processo de refatoração propenso a erros, pois se deve atualizar todas as ocorrências.

Por outro lado, as constantes podem ser referenciadas de muitos lugares, mas precisam ser atualizadas apenas em um único local.

```
+ DBHelper.COLUNA_IDFESTA + " INTEGER PRIMARY KEY, "
+ DBHelper.COLUNA_NOMEFESTA + " TEXT, "
```

```
+ DBHelper.COLUNA_PRECOFESTA + " TEXT, "
+ DBHelper.COLUNA_CRIADORFESTA + " INTEGER, "
+ DBHelper.COLUNA_DESCRICAOFESTA + " TEXT)";
db.execSQL(QUERY_COLUNAFESTA);
}
```

```
String QUERY_COLUNAFESTA = "CREATE TABLE " + DBHelper.TABELA_FESTA + "("
+ DBHelper.COLUNA_IDFESTA + " INTEGER PRIMARY KEY, "
+ DBHelper.COLUNA_NOMEFESTA + " 1 " TEXT, "
```

```

        + DBHelper.COLUNA_PRECOFESTA + " 2 " TEXT, "
        + DBHelper.COLUNA_CRIADORFESTA + " INTEGER, "
        + DBHelper.COLUNA_DESCRICAOFESTA + " TEXT)";
    db.execSQL(QUERY_COLUNAFESTA);
}

```

```

private void criarTabelaUsuario(SQLiteDatabase db) {
    String QUERY_COLUNAUSUARIO = "CREATE TABLE " + DBHelper.TABELA_USUARIO + "("
        + DBHelper.COLUNA_ID + " INTEGER PRIMARY KEY, " + DBHelper.COLUNA_NOME
        + " 3 " TEXT, " + DBHelper.COLUNA_EMAIL + " TEXT, " + DBHelper.COLUNA_SENHA
        + " TEXT)";
    db.execSQL(QUERY_COLUNAUSUARIO);
}

```

Se um campo privado for declarado, mas não usado no programa, ele poderá ser considerado código morto e, portanto, deverá ser removido. Isso melhora a capacidade de manutenção porque os desenvolvedores não se perguntarão para que serve a variável.

```

public class LoginActivity extends AppCompatActivity {
    private EditText editEmail;
    private EditText editSenha;
    private UsuarioServices usuarioServices = new UsuarioServices(this);
    private SessaoUser sessaoUser ;
}

```

Os programadores não devem comentar o código, pois ele incha os programas e reduz a legibilidade. O código não utilizado deve ser excluído e pode ser recuperado do histórico de controle de origem, se necessário.

```

public void onClick(View v) {
    String email = editEmail.getText().toString();
    String senha = editSenha.getText().toString();
    Usuario res = usuarioServices.getUsuario(email, senha);
    if (res != null){
        //sessaoUser.setEmail(email);
    }
}

```

Se um campo privado for declarado, mas não usado no programa, ele poderá ser considerado código morto e, portanto, deverá ser removido. Isso melhora a capacidade de manutenção porque os desenvolvedores não se perguntarão para que serve a variável.

Quando o valor de um campo privado é sempre atribuído aos métodos de uma classe antes de ser lido, ele não está sendo usado para armazenar informações da classe. Portanto, ela deve se tornar uma variável local nos métodos relevantes para evitar qualquer mal-entendido.

```
public class RegisterActivity extends AppCompatActivity {  
    private boolean task = false;
```

```
    private void showExceptionToast(Exception e) {
```

```
        private EditText editNome;  
        private EditText editEmail;  
        private EditText editSenha;  
        private EditText editConfSenha;  
        private Button registrarButton;
```

Se uma variável local for declarada mas não usada, será um código morto e deve ser removida. Isso aumentará a capacidade de manutenção, pois os desenvolvedores não se perguntarão para que serve a variável.

```
    private boolean validarCampos() {  
        String nome = editNome.getText().toString();  
        String email = editEmail.getText().toString();  
        String senha = editSenha.getText().toString();  
        String confSenha = editConfSenha.getText().toString();  
        limparErros();  
        View focusView = null;
```



```
private boolean validarSenha(String senha, String confSenha) {  
    View focusView;
```

```
private boolean validarEmailExiste(String email) {  
    View focusView;
```

```
private boolean validarNomeExiste(String nome) {  
    View focusView;
```

```
private void cadastrar() {  
    String nome = editNome.getText().toString();  
    String email = editEmail.getText().toString();  
    String senha = editSenha.getText().toString();  
    String confSenha = editConfSenha.getText().toString();
```

O retorno de instruções literais booleanas agrupadas em instruções if-then-else deve ser simplificado.

Da mesma forma, as invocações de métodos agrupadas em if-then-else diferentes apenas de literais booleanos devem ser simplificadas em uma única invocação.

```
if (!resultadoValidacoes(nome, email, senha, confSenha)) return false;
```

```
private boolean resultadoValidacoes(String nome, String email, String senha, String  
    if (!validarNomeExiste(nome)) return false;  
    if (!validarEmailExiste(email)) return false;  
    if (!validarSenha(senha, confSenha)) return false;
```

Um armazenamento morto ocorre quando uma variável local recebe um valor que não é lido por nenhuma instrução subsequente. Calcular ou recuperar um valor apenas para substituí-lo ou jogá-lo fora pode indicar um erro grave no código. Mesmo que não seja um erro, é na melhor das hipóteses um desperdício de recursos. Portanto, todos os valores calculados devem ser utilizados.

```
if (senha.isEmpty()) {  
    editSenha.setError("O Campo esta vazio");  
    focusView = editSenha;
```

```
    return false;  
} else if (!validarSenhaIguais(senha, confSenha)) {  
    editSenha.setError("Senhas devem ser iguais");  
    focusView = editSenha;
```

```
if (email.isEmpty()) {  
    editEmail.setError("O Campo esta vazio");  
    focusView = editEmail;
```

```
    return false;  
} else if (!validarEmail(email)) {  
    editEmail.setError("Email inválido");  
    focusView = editEmail;
```

```
if (nome.isEmpty()) {  
    editNome.setError("O campo esta vazio!");  
    focusView = editNome;
```



```

        return false;
    } else if (!validarNome(nome)) {
        editNome.setError("Nome inválido, não aceito caracteres especiais");
        focusView = editNome;
    }
}

```

Complexidade Ciclomática

Complexidade	Avaliação
1-10	Método simples. Baixo risco.
11-20	Método razoavelmente complexo. Moderado risco.
21-50	Método muito complexo. Elevado risco.
51-N	Método de altíssimo risco e bastante instável.






Eventos

Cyclomatic Complexity 58

dominio	14
gui	29
negocio	6
persistencia	9

Gui de eventos

Cyclomatic Complexity 29

 ConsultarEventoActivity.java	2
 CriarEventoActivity.java	3
 EditarEventoActivity.java	3
 ListaEventoActivity.java	5
 RecyclingAdapterFesta.java	16

dominio de eventos

Cyclomatic Complexity 14

 Evento.java	14
---	----

negócio de eventos

Cyclomatic Complexity 6

 EventoServices.java	6
---	---

persistência de eventos

Cyclomatic Complexity 9

 EventoDAO.java	9
--	---

Infra

Cyclomatic Complexity 33

app	2
gui	11
negocio	6
persistencia	14

Infra app

Cyclomatic Complexity 2

SoResenhaApp.java	2
-------------------	---

Infra gui

Cyclomatic Complexity 11

ConfigurationActivity.java	5
MenuActivity.java	4
SplashScreenActivity.java	2

Infra negócio

Cyclomatic Complexity 6

SessaoUsuario.java	4
SoresenhaAppException.java	2

Infra persistência

Cyclomatic Complexity 14

 DBHelper.java	7
 SessaoUser.java	7

Usuário

Cyclomatic Complexity 56

 dominio	11
 gui	33
 negocio	5
 persistencia	7



Usuário domínio

Cyclomatic Complexity 11

 Usuario.java	11
--	----

Usuário gui

Cyclomatic Complexity 33

 LoginActivity.java	5
 RegisterActivity.java	28

Usuário negócio

Cyclomatic Complexity 5

 UsuarioServices.java	5
--	---

Usuário persistência

Cyclomatic Complexity **7**

 UsuarioDAO.java

7