UNIVERSIDADE ESTADUAL DE CAMPINAS

Faculdade de Tecnologia

RICARDO HIDEKI ADATI TOMOMITSU

MELHORIA NA QUALIDADE DE PROJETO JAVA COM SONARQUBE

Análise em projeto legado

LIMEIRA

2021

RICARDO HIDEKI ADATI TOMOMITSU

MELHORIA NA QUALIDADE DE PROJETO JAVA COM SONARQUBE

Análise em Projeto Legado

Monografia apresentada à Faculdade de Tecnologia da Universidade Estadual de Campinas como parte dos requisitos para obtenção do título de Bacharel em Sistemas de Informação

Orientador: Prof. Dr. Plínio Roberto Souza Vilela

Este exemplar corresponde à versão primária da Monografia desenvolvida pelo aluno Ricardo Hideki Adati Tomomitsu e orientada pelo Prof. Dr. Plínio Roberto Souza Vilela.

LIMEIRA

2021

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Faculdade de Tecnologia
Luiz Felipe Galeffi - CRB 8/10385

T598m

Tomomitsu, Ricardo Hideki Adati, 1998-
Melhoria na qualidade de projeto Java com SonarQube : análise em projeto legado / Ricardo Hideki Adati Tomomitsu. – Limeira, SP : [s.n.], 2021.

Orientador: Plínio Roberto Souza Vilela.
Trabalho de Conclusão de Curso (graduação) – Universidade Estadual de Campinas, Faculdade de Tecnologia.

1. Sistemas de informação. 2. Java (Linguagem de programação de computador). 3. Software - Manutenção. I. Vilela, Plínio Roberto Souza,1970-. II. Universidade Estadual de Campinas. Faculdade de Tecnologia. III. Título.

Informações adicionais, complementares

Título em outro idioma: Improvement of Java project with SonarQube: analysis of legacy project
Palavras-chave em inglês:
Information systems
Java (Computer program language)
Software maintenance
Titulação: Bacharel
Banca examinadora:
Plínio Roberto Souza Vilela [Orientador]
Ulisses Martins Dias
Marcos Augusto Francisco Borges
Data de entrega do trabalho definitivo: 12-07-2021

Agradecimentos

Agradeço a todos meus amigos e família, que sempre estiveram ao meu lado, durante toda trajetória do curso.

Ao meu Orientador, Prof. Dr. Plínio Roberto Souza Vilela, por todo apoio providenciado não apenas durante a orientação deste trabalho, mas em todo período lecionado.

À Instituição Unicamp, em especial, os docentes e funcionários que providenciaram tantos conhecimentos e a infraestrutura necessária para o ensino de qualidade, disponibilizado a todos alunos.

Resumo

Este trabalho tem como objetivo verificar a viabilidade na análise e manutenção de um sistema legado, por meio da ferramenta de análise de código SonarQube. Por meio das análises geradas pela ferramenta, foi possível estabelecer os pontos iniciais de análise de código, de acordo com a severidade, camada afetada e tipos de regras afetadas por Bugs, Vulnerabilidades e Code Smells. As análises quantitativa e qualitativa dos resultados obtidos do estudo foram compiladas em tabelas e imagens, a respeito da mudança de códigos e tempo de esforço. Espera-se, com esse trabalho, verificar a viabilidade da ferramenta no tempo de manutenção de um projeto Java já finalizado e adotado em ambiente de produção, assim como verificar o resultado fornecido pela própria ferramenta SonarQube, que pode ser uma ferramenta importante para aprendizado de boas práticas de programação e monitoramento de qualidade de código, dentro ou fora do ambiente acadêmico.

Abstract

This work objective is to verify the feasibility of analyzing and maintaining a legacy system, using the SonarQube code analysis tool. Through the analysis generated by the tool, it was possible to establish the starting points of code analysis, according to severity, affected layer and types of rules affected by Bugs, Vulnerabilities and Code Smells. Quantitative and qualitative analyzes of the results obtained from the study were compiled in tables and images, regarding code change and effort time. It is expected, with this work, to verify the feasibility of the tool in the maintenance time of a Java project already completed and adopted in the production environment, as well as verifying the result provided by the SonarQube tool itself, which can be an important tool for learning good programming practices and code quality monitoring, inside or outside the academic environment.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Este capítulo tem como objetivo introduzir o leitor ao projeto, contendo tópicos que descrevem o contexto do projeto, sua motivação, seu objetivo e a estrutura do texto da monografia desenvolvida.

1.1. Contexto

Vivemos em uma era digital na qual cada vez mais sistemas são desenvolvidos para suprir as necessidades de milhares de pessoas e empresas, das mais diversas áreas. O surgimento constante de novos sistemas acaba por gerar a competitividade, que por consequência, gera a necessidade da inovação e melhoria na qualidade de serviços e produtos. Essa necessidade está diretamente ligada aos atributos essenciais de um bom software: a Confiança, Manutenibilidade, Eficiência e Aceitabilidade (SOMMERVILLE, 2011). Por conta disso, manter um código bem estruturado, o que leva em consideração a necessidade de uma boa legibilidade, segurança e uso otimizado de recursos, passou a ser um pré-requisito para as empresas e desenvolvedores, para garantir um sistema com boa capacidade de manutenção e apto para acompanhar e atender às constantes mudanças de demandas dos mais diversos usuários.

A Manutenibilidade é um conceito que descreve a necessidade do desenvolvimento de um código que possa ser adaptado às necessidades dos clientes, e é um dos atributos mais críticos no desenvolvimento de Software, visto que os sistemas e ambientes de negócios passam por constantes mudanças, seja pelo surgimento de novas tecnologias ou pela própria necessidade do cliente.

1.2. Motivação

Na Engenharia de Software, a manutenção do software tem início a partir do momento que o sistema desenvolvido é entregue para uso. De acordo com Shari Pfleeger (2007), a manutenção de software pode ser dividida em 4 tipos:

- Manutenção Corretiva;
- Manutenção Adaptativa;
- Manutenção Perfectiva;
- Manutenção Preventiva;

Apesar de sua extrema importância, o processo de manutenção de um software possui vários fatores que podem tornar o processo complexo, portanto, deve ser realizado de maneira organizada para que se torne viável durante o ciclo de vida de um sistema. Alguns estudos e pesquisas, como a de Guimarães (1983), indicam que o custo de manutenção pode equivaler ou superar muito o custo do próprio desenvolvimento do software.

A dificuldade na manutenção do código está diretamente relacionada à estrutura do projeto e das práticas de programação adotadas durante a etapa de desenvolvimento do sistema. Manter uma previsão de manutenção é de extrema importância para monitorar os custos gerados durante essa etapa, porém, a manutenção está diretamente ligada à quantidade e complexidade de interfaces do sistema, e pode necessitar uma análise mais complexa do código, antes de ser estabelecida.

O SonarQube, por ser uma ferramenta focada na análise contínua do código e fornecendo métricas para análise e estimativas de esforço para melhoria e correção de problemas, pode exercer grande importância nesse processo.

1.3. Objetivo

O objetivo deste projeto é aplicar a análise do SonarQube e mostrar sua viabilidade no processo de manutenção de código para um novo desenvolvedor (que não participou do desenvolvimento inicial do sistema), levando em consideração pontos quantitativos e qualitativos de análises do código, assim como a possibilidade do uso da ferramenta, no apoio da previsão de tempo e custo de manutenção de um projeto legado.

1.4. Estrutura do texto

Este documento apresenta, na forma de relatório técnico, os resultados da aplicação da ferramenta SonarQube, na manutenção e análise de qualidade de Código Java, do sistema escolar “Administração Escolar People”, desenvolvido pelo Prof. Dr. Plínio Roberto Souza Vilela.
O relatório foi dividido em capítulos, contendo análises e informações obtidas ao longo do projeto:

- Capítulo 2 – Metodologia: descrição da metodologia utilizada no projeto;
- Capítulo 3 – Fundamentação teórica, descrição do sistema e SonarQube: descrição de estudos teóricos, do sistema e do SonarQube;
- Capítulo 4 - Resultados: resultados da análise, com foco na etapa de manutenção do código e respectivas análises levantadas pela ferramenta SonarQube.
- Capítulo 5 – Conclusão: conclusão final do trabalho.
- Capítulo 6 – Referências e Apêndice A: referencias bibliográficas e descrição de possíveis soluções das regras infringidas, durante a análise do SonarQube.

Capítulo 2

Este capítulo tem como objetivo descrever a metodologia adotada para o desenvolvimento do projeto.

Metodologia

A metodologia aplicada no projeto se baseia nas seguintes etapas, estabelecidas no início do projeto e executadas ao longo dele:

- Reuniões (Orientador e Aluno): alinhamento a respeito dos objetivos e próximos passos do projeto, ao longo do semestre vigente.
- Estudo da ferramenta SonarQube: verificação da viabilidade da aplicação da ferramenta, no contexto do projeto.
- Execução inicial do SonarQube: análise inicial do projeto, alvo do estudo.
- Documentação dos problemas e dados levantados pelo SonarQube: documentação de dados importantes no contexto do projeto (Esforço estimado, quantidade de problemas encontrados)
- Validação de problemas encontrados pelo SonarQube e Manutenção de código: estudo dos problemas encontrados, e possíveis mudanças de código, além da documentação de esforços gastos.
- Análise de tempos de Manutenção do Código: estudo do esforço gasto ao longo do projeto.
- Análise Final do SonarQube e Documentação de Resultados e Aprendizados: considerações finais, a respeito da viabilidade do uso da ferramenta.

Capítulo 3

Este capítulo tem como objetivo fornecer ao leitor uma descrição dos estudos realizados a respeito do contexto do projeto, assim como explicar alguns fundamentos importantes para o entendimento de seus resultados. O capítulo contém tópicos que descrevem a fundamentação teórica do projeto, o histórico e características do sistema alvo, o funcionamento e termos do SonarQube, e uma breve descrição das análises iniciais do código do sistema analisado.

Fundamentação Teórica

A qualidade de Software está interligada aos atributos de Capacidade de Manutenção, Confiança e Proteção, Eficiência e Aceitabilidade (SOMMERVILLE, 2011).

Os atributos de “Capacidade de Manutenção” e “Confiança e Proteção” estão diretamente relacionados a uma boa qualidade de código, visto que estão relacionados com a segurança e capacidade de evolução do código, de acordo com as necessidades dos clientes.

Através de estudos empíricos a respeito da mudança de sistemas, Lehman propôs 8 leis, a respeito da evolução dos softwares (LEHMAN,1997). As leis são:

- Mudança Contínua
- Aumento da Complexidade
- Evolução de programa de grande porte
- Estabilidade organizacional
- Conservação de familiaridade
- Crescimento contínuo
- Declínio de qualidade
- Sistema de feedback

Dentre as 8 leis, destaca-se as leis da mudança contínua, aumento da complexidade e declínio de qualidade, no contexto da manutenção de software.

A Mudança Contínua descreve que programas que são implementados em ambientes do mundo real, precisam mudar, caso contrário, se tornam menos úteis no ambiente.

O Aumento da Complexidade descreve que com a mudança de um programa, há a tendência de que a complexidade do código se torne maior e, por conta disso, é necessário dedicar esforços para preservar e simplificar a estrutura.

O Declínio de Qualidade descreve que a qualidade do sistema tende a cair, caso não seja modificado para se adaptar às mudanças do ambiente operacional.

A manutenção do Software, que começa a partir da liberação de uso do sistema, e o controle de qualidade, que envolve séries de inspeções, revisões e testes adotados para garantir que os requisitos sejam atendidos (PRESSMAN, 2006), passam então, a exercerem um papel crucial para manter um código ideal.

Ao realizar a manutenção de um sistema legado, é necessário levar em consideração vários fatores, antes de determinar se é um processo viável. De acordo com Sommerville (2011) há 4 estratégias que podem ser adotadas durante o gerenciamento de um sistema legado:

- Descartar completamente o sistema
- Não mexer no sistema, e continuar com uma manutenção regular
- Reestruturar o sistema para melhorar a manutenibilidade
- Substituir parte ou totalidade do sistema, por um novo sistema

Ao realizar análises de código, o SonarQube leva em consideração as seguintes regras, para identificar e gerar os problemas na plataforma, conforme a figura 3.1:

- **Code Smell** (Domínio de Capacidade de Manutenção): de acordo com o SonarQube, é um problema que faz o código confuso e dificulta sua manutenibilidade;
- **Bug** (Domínio de Confiabilidade): de acordo com o SonarQube, trata-se de um erro de código que precisa ser corrigido imediatamente;

- **Vulnerability** (Domínio de Segurança): de acordo com o SonarQube, é um ponto do código que está vulnerável à ataques;
- **Security Hotspot** (Domínio de Segurança): de acordo com o SonarQube, é uma possível falha de segurança em um trecho de código. Deve ser revisado, para confirmar se pode ser ignorado, ou se realmente se trata de uma falha;

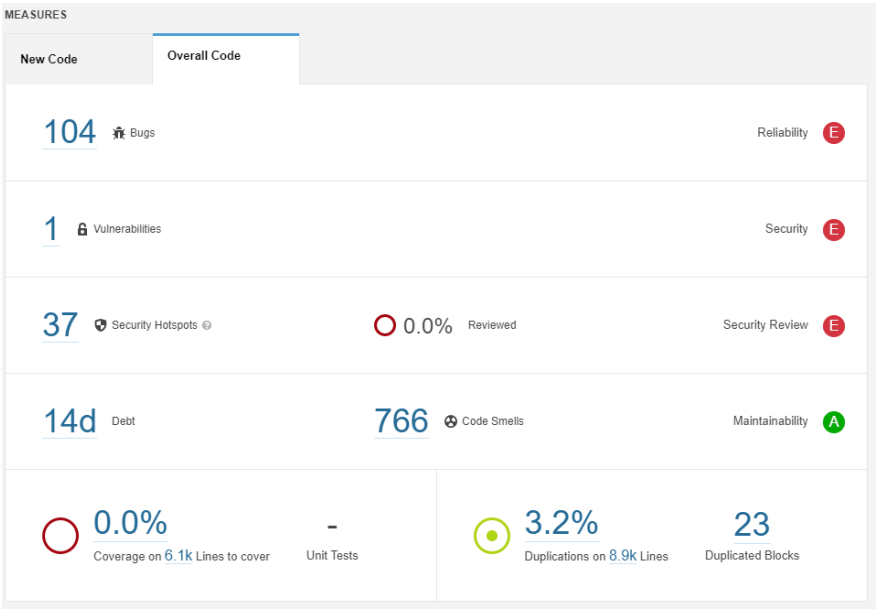


Figura 3.1 – Dashboard Inicial do Projeto SonarQube

O SonarQube também disponibiliza uma tabela, conforme a tabela 3.1, que disponibiliza as informações de cada pacote e classe do projeto, possibilitando a checagem das quantidades e tipos de erros encontrados ao longo do projeto.

	Lines of Code	Bugs	Vulnerabilities	Code Smells	Security Hotspots	Coverage	Duplications
src/main/java	8,837	104	1	766	37	0.0%	3.3%
control	462	0	0	32	0	0.0%	0.0%
model	3,389	102	1	166	37	0.0%	5.2%
utils	81	0	0	5	0	0.0%	0.0%
view	4,905	2	0	563	0	0.0%	2.3%

Tabela 3.1 – Tabela Geral SonarQube

Com essas informações em mãos, é possível verificar e validar os problemas encontrados e realizar as devidas tratativas.

O sistema

O sistema alvo das análises de código é o sistema “Administração Escolar People”. Desenvolvida pelo Professor Doutor Plínio Roberto Souza Vilela. Possui como escopo a organização das atividades escolares de uma instituição de ensino de Inglês e Computação, e foi utilizado por uma instituição de ensino (Ambiente de Produção). A figura 3.3, disponibilizada na próxima página se trata da tela inicial do sistema.

O Commit inicial do projeto foi feito no BitBucket, em Julho de 2015, e o último commit foi realizado em Novembro de 2018.

O projeto contém 8837 linhas de código e se baseia na programação orientada a objetos, seguindo a arquitetura MVC (Model, Controller, View). Também contém classes DAO desenvolvidas manualmente e possui um banco de dados local MySql.

O uso do padrão MVC, demonstrado na figura 3.2, tem como intuito principal o isolamento do código do projeto em camadas específicas. De maneira geral, isso faz com que a camada lógica e a interface de usuário sejam isoladas entre si, facilitando a refatoração e o reuso de classes, assim como a manutenibilidade do código, em caso de futuras mudanças. As 3 camadas são:

- Model: Camada que gerencia os dados do sistema e seu armazenamento persistente.
- Controller: Camada que intermedia a camada View e Model, além de implementar as regras de negócio do sistema.
- View: Camada à qual pertence as interfaces gráficas, direcionadas à interação do usuário do sistema.

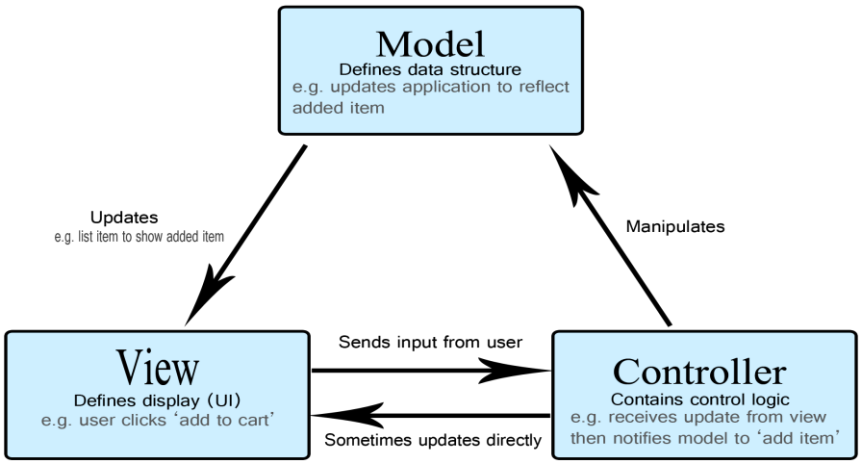


Figura 3.2 – Arquitetura MVC (Obtida em: https://developer.mozilla.org/em-US/docs/Glossary/MVC)

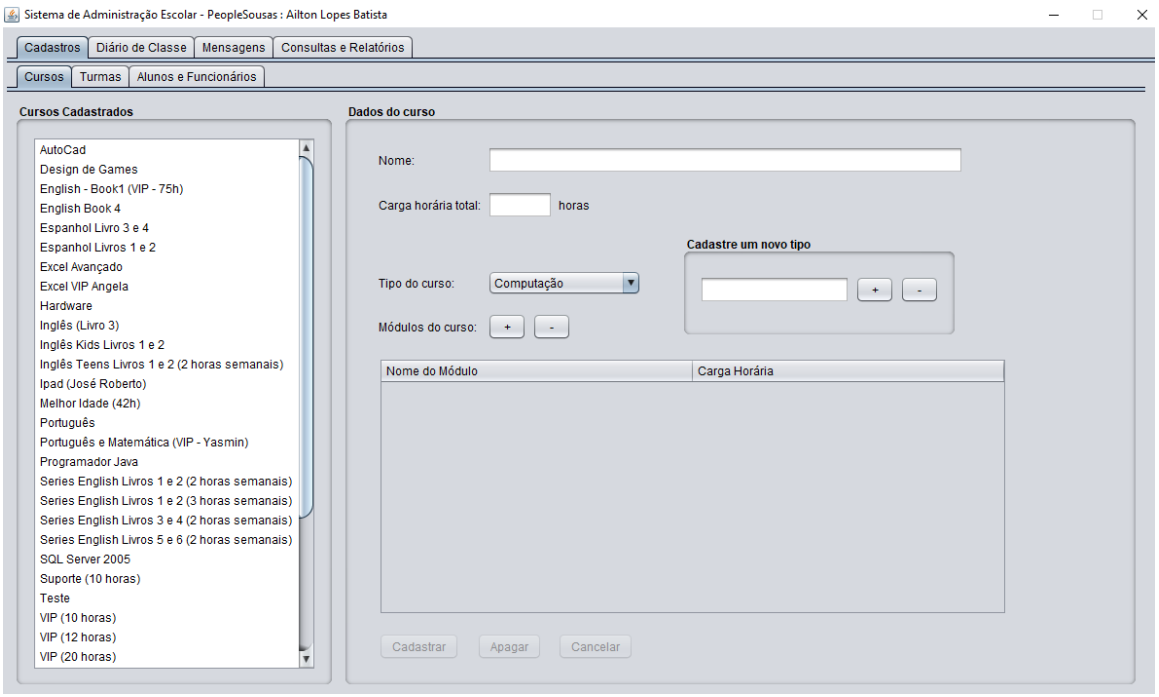


Figura 3.3 – Tela principal do Sistema

SonarQube

SonarQube é uma ferramenta automática de análise de códigos, para detectar problemas no código. Conforme a tabela 3.2, o SonarQube divide e define os problemas encontrados em 4 tipos:

- **Bug:** Um erro de código que precisa ser corrigido imediatamente
- **Vulnerabilidade:** Um ponto do código que está vulnerável à ataques
- **Code Smell:** Problema que faz o código confuso e dificulta sua manutenibilidade.
- **Security Hotspot:** Possível falha de segurança em um trecho de código. Deve ser revisado, para confirmar se pode ser ignorado, ou se realmente é uma falha.

As regras infringidas também são divididas em 5 graus de severidade:

- **Blocker:** um problema encontrado, com alta probabilidade de impactar o funcionamento da aplicação em produção. O código deve ser arrumado imediatamente
- **Critical:** um problema encontrado, com uma baixa probabilidade de afetar o funcionamento da aplicação em produção, ou um problema que representa falhas de segurança.
- **Major:** falha de qualidade que pode impactar muito a produtividade do desenvolvedor.
- **Minor:** falha de qualidade que pode afetar um pouco a produtividade do desenvolvedor.
- **Info:** não é um problema encontrado e nem uma falha de qualidade, mas sim, apenas um achado.

O SonarQube disponibiliza uma estimativa de tempo para correção de cada regra infringida, levando em consideração sua severidade, tipo de regra e suas possíveis correções. Isso possibilita uma visão mais adequada, do quanto de esforço uma melhoria ou correção do código podem gerar. O SonarQube é uma ferramenta capaz de prover ao usuário, um robusto sistema, para monitoramento da qualidade do código, e estabelecer possíveis estimativas de esforço a serem gastos ao longo do código, assim como documentar os esforços, durante o ciclo de manutenção.

Package	Bug	Code Smell	Security Hotspot	Vulnerability	Total
Control		**32**			**32**
Critical		1			**1**
Major		5			**5**
Minor		26			**26**
Model	**102**	**166**	**37**	**1**	**306**
Blocker	102			1	**103**
Critical		10			**10**
High			22		**22**
Low			15		**15**
Major		47			**47**
Minor		109			**109**
Utils		**5**			**5**
Critical		1			**1**
Major		2			**2**
Minor		2			**2**
View	**2**	**563**			**565**
Critical		40			**40**
Info		2			**2**
Major	1	310			**311**
Minor	1	211			**212**
Total	**104**	**766**	**37**	**1**	**908**

Tabela 3.2 – Tipos de Regras Infringidas por Camada do código

Ao selecionar a regra infringida, o SonarQube permite explorar as linhas de código com o problema citado, destacando no código, conforme exemplo da figura 3.5. Isso facilita a localização das linhas de código com cada problema encontrado, além das classes em que se encontram os problemas detectados.

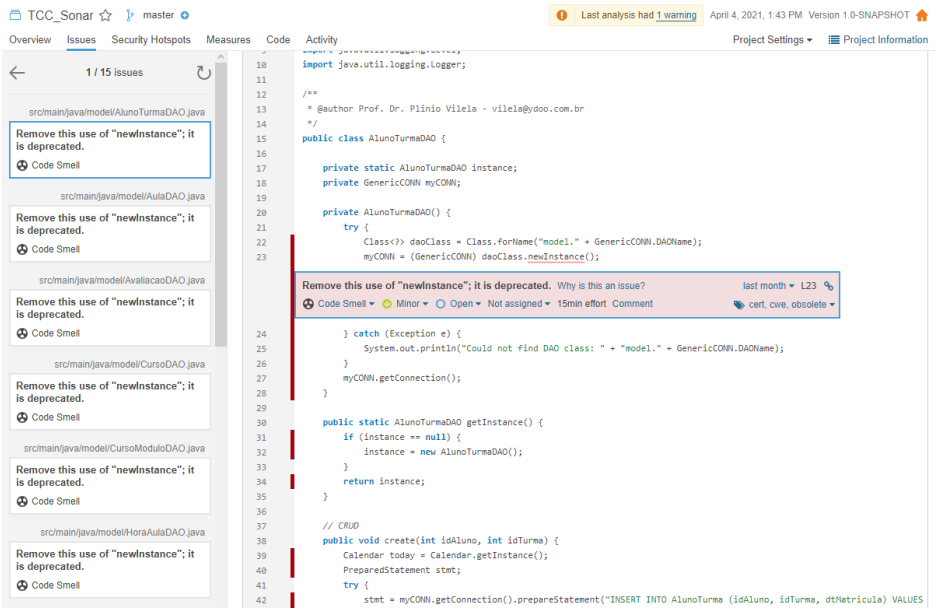


Figura 3.5 – Localização da regra infringida

O Sonarqube também providencia uma documentação, com as descrições das regras infringidas, e exemplos de códigos não compatíveis e compatíveis, para facilitar na resolução dos problemas encontrados, conforme a figura 3.6. Além disso, o SonarQube permite verificar o histórico de mudanças. Usuários podem comentar cada “análise” feita, e gerar versões, conforme a figura 3.7.

Noncompliant Code Example

```
private void readTheFile() throws IOException {
  Path path = Paths.get(this.fileName);
  BufferedReader reader = Files.newBufferedReader(path, this.charset);
  // ...
  reader.close();  // Noncompliant
  // ...
  Files.lines("input.txt").forEach(System.out::println); // Noncompliant: The stream needs to be closed
}

private void doSomething() {
  OutputStream stream = null;
  try {
    for (String property : propertyList) {
      stream = new FileOutputStream("myfile.txt");  // Noncompliant
      // ...
    }
  } catch (Exception e) {
    // ...
  } finally {
    stream.close();  // Multiple streams were opened. Only the last is closed.
  }
}
```

Compliant Solution

```
private void readTheFile(String fileName) throws IOException {
    Path path = Paths.get(fileName);
    try (BufferedReader reader = Files.newBufferedReader(path, StandardCharsets.UTF_8)) {
      reader.readLine();
      // ...
    }
    // ..
    try (Stream<String> input = Files.lines("input.txt"))  {
      input.forEach(System.out::println);
    }
}

private void doSomething() {
  OutputStream stream = null;
  try {
    stream = new FileOutputStream("myfile.txt");
    for (String property : propertyList) {
      // ...
    }
  } catch (Exception e) {
    // ...
  } finally {
    stream.close();
  }
}
```

Figura 3.6 – Exemplos: Código não compatível e compatível

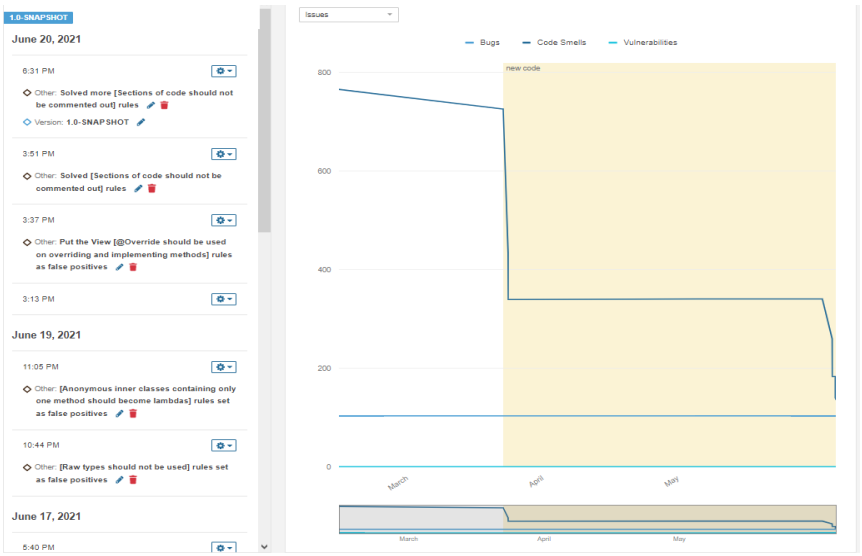


Figura 3.7 – Gráfico de atividade SonarQube

Atualmente, para a análise da linguagem JAVA, o SonarQube possui Scanners compatíveis com projetos Gradle e Maven. Para dar início às análises do projeto, foi necessário migrar o projeto Java para um projeto “Maven”. Foi criado um projeto Maven, no IDE Netbeans, e as classes foram exportadas ao mesmo. Após isso, foram revisadas e importadas as dependências Javas ausentes no projeto, conforme a figura 3.4.

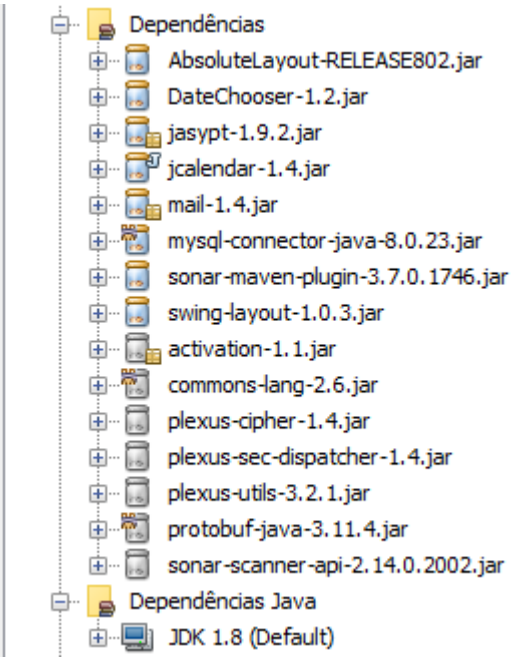


Figura 3.4 – Dependências Java

Com o projeto funcional, foi feita a instalação e execução da instância local do SonarQube, além da configuração do XML maven.

Por padrão, o acesso ao SonarQube é feito localmente, através do login em http://localhost:9000/. O acesso ao repositório se encontra a seguir: https://bitbucket.org/pvilela/admclassroom/src/version0.9/

Por meio das análises constantes da ferramenta, é possível verificar e acompanhar a qualidade e segurança do código.

Análise inicial do Projeto

Inicialmente, ao executar a análise do projeto no SonarQube, foram estabelecidos conforme a tabela 3.3:

Package	Bugs	Code Smells	Security Hotspots	Vulnerabilities
Model	102	166	37	1
Control	0	32	0	0
Utils	0	5	0	0
View	2	563	0	0
Total	**104**	**766**	**37**	**1**

Tabela 3.3 – Tipos Infringidos X Quantidade X Camada

Conforme analisado, a maioria dos code smells se encontra na camada View, porém, por conter principalmente a interface gráfica do sistema, ela não será o foco de análise atual.

Além disso, grande parte dos code smells contidos na camada View, se basearam em códigos automaticamente gerados (principalmente as variáveis criadas automaticamente para interface utilizadas em botões e campos de texto), e foram descartados da análise, conforme será explicado nos próximos tópicos do projeto.

Dessa forma, a análise foi direcionada principalmente para a camada Model, que por sua vez contém a maior parte das classes, e é responsável pelo gerenciamento dos dados e armazenamento persistente do sistema.

Por meio das análises de código do SonarQube, foi documentado todos os tipos de code smells, bugs, vulnerabilidades e falhas de segurança.

Capítulo 4

Este capítulo contém os resultados obtidos ao longo das análises e do desenvolvimento do projeto, descritos ao longo do tópico “Resultados”.

Resultados

Após análise inicial do código do projeto, foi realizado a documentação das regras infringidas de cada tipo e severidade, assim como as descrições deles, conforme o apêndice A. Além da descrição, também foi realizado mudanças nos códigos, de acordo com as recomendações do SonarQube.

De acordo com Shari Pfleeger (2007), ao medir o tempo médio de correção de problemas, é necessário levar em consideração a documentação dos seguintes tempos:

- Tempo utilizado para analisar o problema
- Tempo necessário para especificar as mudanças a serem feitas
- Tempo necessário para realizar as mudanças
- Tempo necessário para testar as mudanças
- Tempo necessário para documentar as mudanças.

Por fornecer uma estimativa de tempo a ser utilizado para solução das regras infringidas durante a análise do código, além de localizar os trechos de código responsáveis pela violação de regra, o SonarQube possibilitou identificar as camadas mais afetadas, assim como definir as severidades das regras infringidas, contidas no “Apêndice A” da monografia.

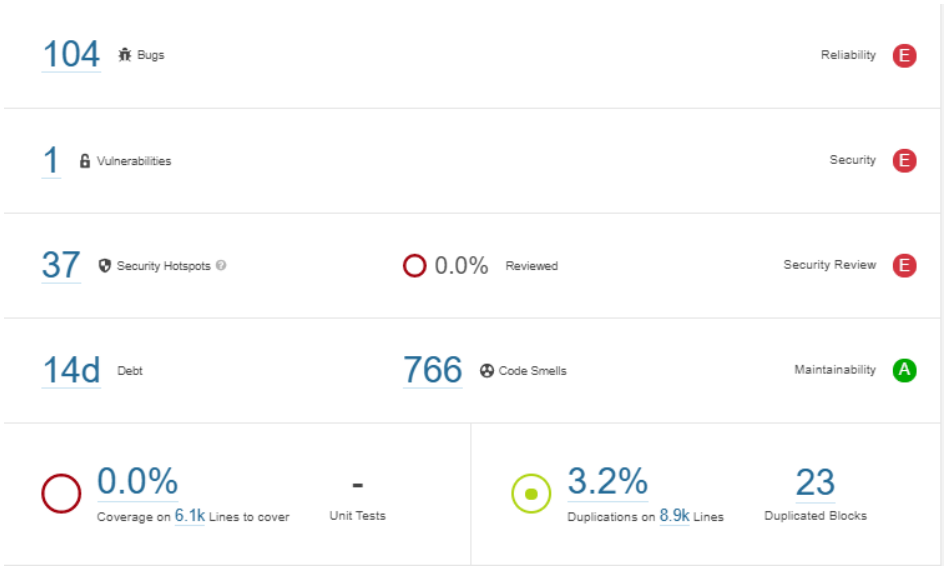


Figura 4.1 – Dashboard inicial SonarQube

Conforme a figura 4.1, o SonarQube estimou 14 dias de débito, para resolução dos problemas encontrados. O tempo de débito representa o tempo faltante, para solução dos code smells encontrados no código, e considera que

a carga horária por dia é de 8 horas (ou seja, haveria aproximadamente 112 horas de esforço).

Os tempos estimados por regra são calculados em minutos, e foram documentados em uma planilha, em conjunto com o esforço gasto durante a mudança de códigos. As prioridades para manutenção do código se basearam na severidade apontada pelo SonarQube, nos atributos que afetam (Manutenibilidade, Confiabilidade e Segurança) e nas camadas afetadas.

Os 14 dias de débito ficaram divididos entre 4 tipos de problemas encontrados pelo SonarQube, conforme a tabela 4.1, sendo eles:

- Code Smells
- Bugs
- Vulnerabilidades
- Security Hotspots

Os security Hotspots são considerados pontos a serem validados pelo próprio desenvolvedor, a fim de determinar se são realmente problemas, ou se podem ser ignorados na análise, portanto, não possuem uma estimativa prévia, de tempo de solução.

Type	Average Total Effort (Time in minutes) Preview
Code Smell	6803
Bug	525
Vulnerability	45
Security Hotspot	0
Total	**7373**

Tabela 4.1 – Tempo estimado de esforço por tipo

Code Smells

Os tempos estimados (fornecidos pelo SonarQybe) para correção dos Code Smells foram distribuídos conforme a figura 4.2.

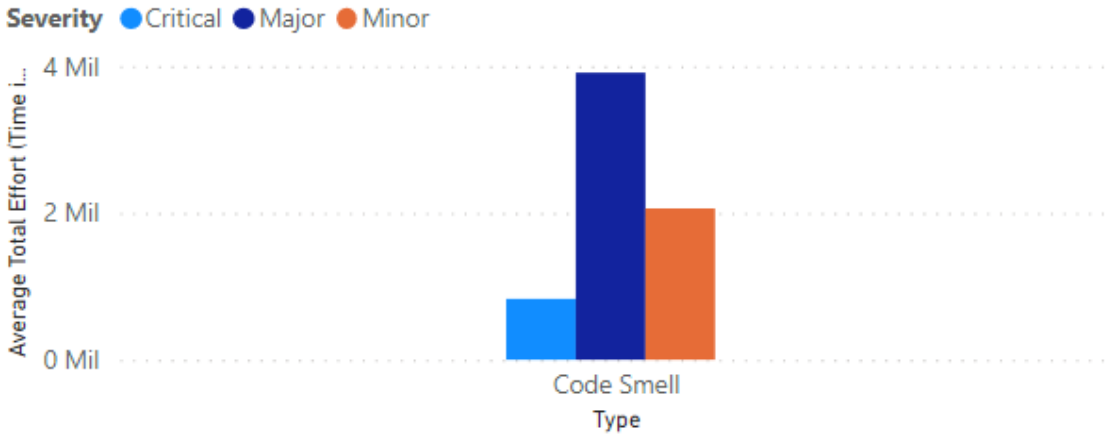


Figura 4.2 – Estimativa de esforço X Severidade (Code Smell)

A maior parte dos Code Smells detectados pelo SonarQube remetem-se à severidade “Major”. As camadas mais afetadas do projeto foram a camada View e Model, conforme consta na figura 4.3.

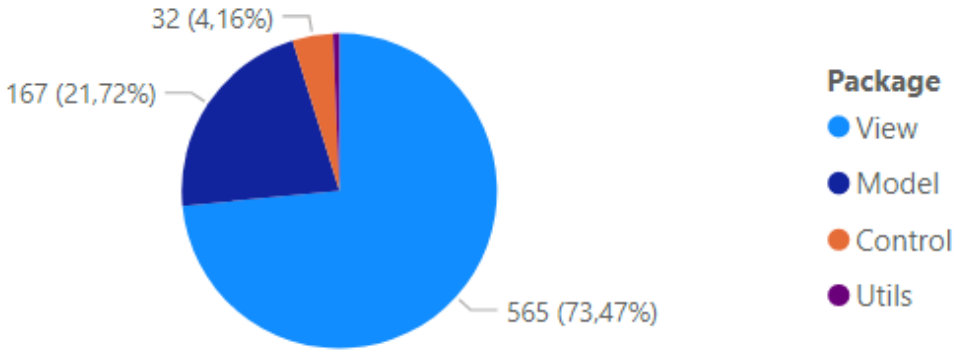


Figura 4.3 – Quantidade X Camada (Code Smell)

A camada View está diretamente relacionada à interface do sistema. Por conta disso, foi feito uma análise inicial de tal camada, visto que grande parte dos códigos teriam sido gerados automaticamente. Analisando as regras infringidas nessa camada, foi possível perceber que várias delas apontaram como erros, códigos gerados automaticamente, como no momento da criação de botões, listas e ações de clique. Tais erros foram eliminados da análise, considerados como falsos-positivos. Dessa forma, cerca de 434 Code Smells foram eliminados, sendo os mesmos de severidade Major e Minor, conforme demonstrado nas figura 4.4 e na tabela 4.2. Tais Code Smells eram responsáveis por cerca de 29% do tempo total estimado pelo SonarQube, equivalendo a cerca de 36 horas de trabalho.

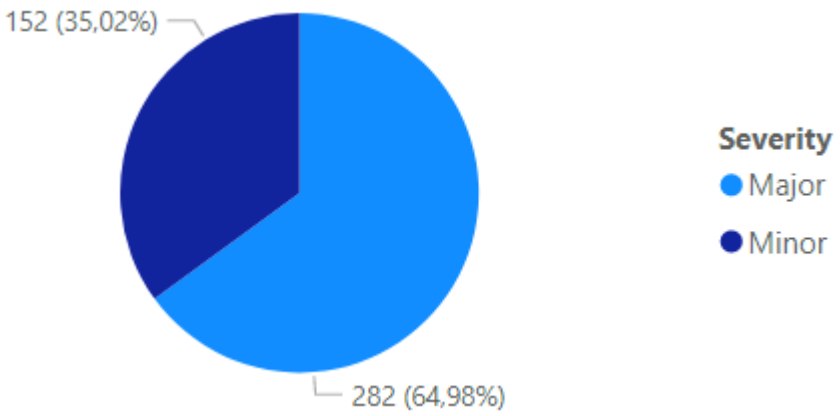


Figura 4.4 – Quantidade de Code Smells “ignorados” X Severidade

Rule	Type	Average Total Effort (Time in minutes) Preview	Severity
Private fields only used as local variables in methods should become local variables	Code Smell	760	Minor
Unused method parameters should be removed	Code Smell	475	Major
Raw types should not be used	Code Smell	410	Major
Anonymous inner classes containing only one method should become lambdas	Code Smell	380	Major
@Override should be used on overriding and implementing methods	Code Smell	145	Major
Total		**2170**	

Tabela 4.2 – Code Smells excluídos da análise

Dentre os Code Smells de severidade “Minor”, descritos na tabela 4.3, o tempo estimado pelo SonarQube, para correção de cada ocorrência, variou de 1 a 15 minutos. Tais regras violadas se basearam principalmente em boas práticas de programação.

Rule	Type	Severity	Quantity	Average Total Effort (Time in minutes) Preview
@Deprecated code should not be used	Code Smell	Minor	16	240
Array designators "[]" should be on the type, not the variable	Code Smell	Minor	6	30
Boolean literals should not be redundant	Code Smell	Minor	6	30
Boxed "Boolean" should be avoided in boolean expressions	Code Smell	Minor	3	15
Catches should be combined	Code Smell	Minor	18	90
Class variable fields should not have public accessibility	Code Smell	Minor	1	10
Declarations should use Java collection interfaces such as "List" rather than specific implementation classes such as "LinkedList"	Code Smell	Minor	76	760
Local variables should not be declared and then immediately returned or thrown	Code Smell	Minor	1	2
Multiple variables should not be declared on the same line	Code Smell	Minor	2	4
Private fields only used as local variables in methods should become local variables	Code Smell	Minor	152	760
Public constants and fields initialized at declaration should be "static final" rather than merely "final"	Code Smell	Minor	2	4
Redundant casts should not be used	Code Smell	Minor	5	25
Return of boolean expressions should not be wrapped into an "if-then-else" statement	Code Smell	Minor	6	12
Static non-final field names should comply with a naming convention	Code Smell	Minor	1	2
switch statements should have at least 3 "case" clauses	Code Smell	Minor	1	5
The diamond operator ("<>") should be used	Code Smell	Minor	40	40
throws declarations should not be superfluous	Code Smell	Minor	1	5
Unnecessary imports should be removed	Code Smell	Minor	10	18
Unused local variables should be removed	Code Smell	Minor	2	10
Total			**349**	**2062**

Tabela 4.3 – Minor Code Smells X Quantidade e esforço

Dentre os Code Smells de severidade “Major”, contidos na tabela 4.4, o tempo estimado para correção variou de 5 minutos a 270 minutos para cada ocorrência de regra violada. A alta divergência de tempos estimados ocorre por conta das regras infringidas estarem ligadas à necessidade de revisar diretamente a estrutura do código, como por exemplo, a árvore de herança complexa e a duplicação de linhas de códigos.

Type	Rule	Severity	Quantity	Average Total Effort (Time in minutes) Preview
Code Smell	Unused method parameters should be removed	Major	95	475
Code Smell	Raw types should not be used	Major	82	410
Code Smell	Anonymous inner classes containing only one method should become lambdas	Major	76	380
Code Smell	@Override should be used on overriding and implementing methods	Major	29	145
Code Smell	Sections of code should not be commented out	Major	25	125
Code Smell	Standard outputs should not be used directly to log anything	Major	19	190
Code Smell	Source files should not have any duplicated blocks	Major	11	340
Code Smell	Inheritance tree of classes should not be too deep	Major	6	1620
Code Smell	Constructors should not be used to instantiate "String", "BigInteger", "BigDecimal" and primitive-wrapper classes	Major	3	15
Code Smell	Methods should not have too many parameters	Major	3	60
Code Smell	Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used	Major	3	60
Code Smell	Collapsible "if" statements should be merged	Major	2	10
Code Smell	Nested blocks of code should not be left empty	Major	2	10
Code Smell	A field should not duplicate the name of its containing class	Major	1	10
Code Smell	Constructors of an "abstract" class should not be declared "public"	Major	1	2
Code Smell	Generic exceptions should never be thrown	Major	1	20
Code Smell	Local variables should not shadow class fields	Major	1	5
Code Smell	Presenca.java	Major	1	10
Code Smell	TurmaHorario.java	Major	1	5
Total			**364**	**3912**

Tabela 4.4 – Major Code Smells X Quantidade e esforço

Dentre os Code Smells de severidade “Critical”, contidos na tabela 4.5, o tempo estimado de correção apontado pelo SonarQube variou de 2 a 30 minutos, para cada ocorrência. As violações de regras apontadas estão diretamente relacionadas à legibilidade e manutenibilidade do código, visto que envolve pontos que que afetam o processo de refatoração e o entendimento do próprio código.

Type	Rule	Severity	Quantity	Average Total Effort (Time in minutes) Preview
Code Smell	String literals should not be duplicated	Critical	18	180
Code Smell	Fields in a "Serializable" class should either be transient or serializable	Critical	17	510
Code Smell	Constant names should comply with a naming convention	Critical	5	6
Code Smell	switch statements should have "default" clauses	Critical	4	20
Code Smell	Instance methods should not write to "static" fields	Critical	3	60
Code Smell	Methods should not be empty	Critical	3	15
Code Smell	static base class members should not be accessed via derived types	Critical	3	15
Code Smell	Cognitive Complexity of methods should not be too high	Critical	1	23
Total			**54**	**829**

Tabela 4.5 – Critical Code Smells X Quantidade e esforço

O tempo estimado condiz com o tempo necessário para resolver os problemas apontados no código, porém, a repetição das regras ao longo do código faz com que sua manutenção seja mais prática, nas demais ocorrências, tendo em vista que o motivo do problema e como resolver o mesmo, já foram aprendidos pelo novo desenvolvedor.

Bugs

Conforme demonstrado na figura 4.5, a análise inicial do SonarQube levantou 104 bugs no sistema, divididos entre as severidades Blocker, Major e Minor.

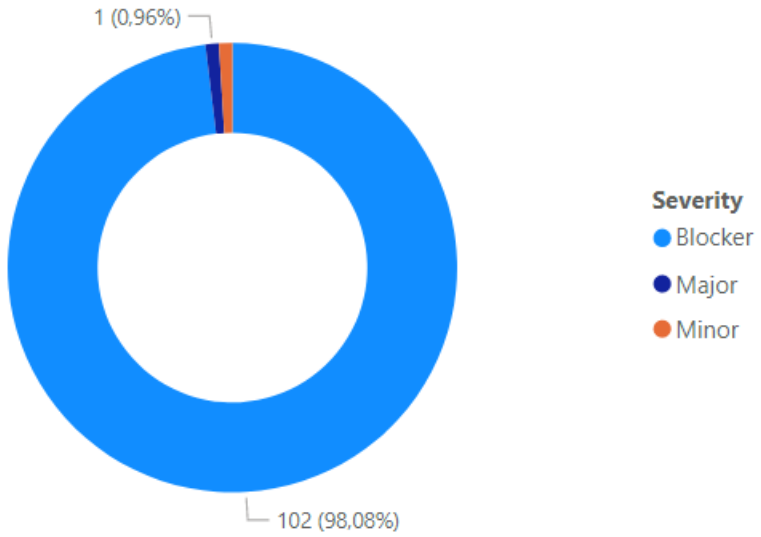


Figura 4.5 – Quantidade de bugs X Severidade

A distribuição dos bugs por camada consta na figura 4.6:

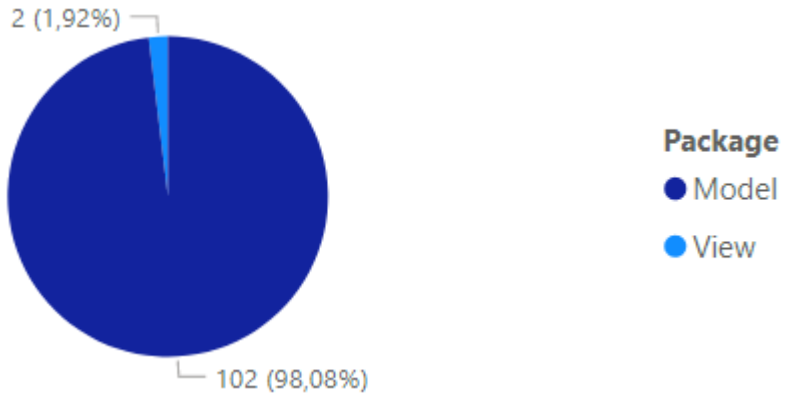


Figura 4.6 – Quantidade de Bugs X Camada

A maioria dos Bugs foram apontados principalmente nas classes DAO, da camada Model, e se referem à necessidade de fechar conexões realizadas, principalmente ao banco de dados. A descrição dos problemas descritos na tabela 4.6 se encontram no apêndice A da monografia

Type	Rule	Severity	Quantity	Average Total Effort (Time in minutes) Preview
Bug	Math operands should be cast before assignment	Minor	1	5
Bug	Null pointers should not be dereferenced	Major	1	10
Bug	Resources should be closed	Blocker	102	510
Total			**104**	**525**

Tabela 4.6 – Bugs X Severidade e estimativa de esforço

Dentre os bugs encontrados, o SonarQube estimou de 5 a 10 minutos para correção de cada bug, o que condiz com o tempo médio gasto para correção da regra com maior quantidade, contidos na tabela 4.7.

Rule	Type	Severity	Average Total Effort (Time in minutes) Spent	Quantity
Resources should be closed	Bug	Blocker	408	102

Tabela 4.7 – Esforço gasto X Bug

Vulnerabilities

A análise do SonarQube apenas apresentou 1 vulnerabilidade. Sendo ela relacionada à regra “Databases should be password-protected”, que indica que o banco de dados não está protegido. Sua descrição se encontra no apêndice.

Security Hotspots

O SonarQube levantou 37 possíveis falhas de segurança, que precisam ser analisadas.

Severity	Type	Quantity
High	Authentication	2
High	SQL Injection	19
Low	Insecure Configuration	16
Total		**37**

Tabela 4.8 – Tipo de Security Hotspot X Quantidade e tipo

Além da severidade, o SonarQube também separou as regras em 3 tipos, conforme a tabela 4.8.

Todas as regras infligidas se remetem às conexões realizadas ao Banco de dados local, e estão relacionadas à presenças de senhas de acesso ao banco de dados no código, Injeções SQL e funcionalidades de debug ainda presentes no código final.

O aumento da quantidade de dados gerados pelos diversos sistemas utilizados no cotidiano das pessoas teve como consequência o aumento global na importância da segurança de dados. Com a Lei Geral de Proteção de Dados (LGPD), aprovada em 2018, garantir a segurança dos dados e do sistema passou a ser um pré-requisito para as empresas e dos mais diversos serviços providenciados. Com essa visão, garantir que o acesso à informação esteja sempre segura é de extrema importância, e o SonarQube possibilita a análise de possíveis detratores de segurança, contido no projeto.

Após a análise e exclusão de problemas de código, a análise final levantou as informações contidas na figura 4.7.

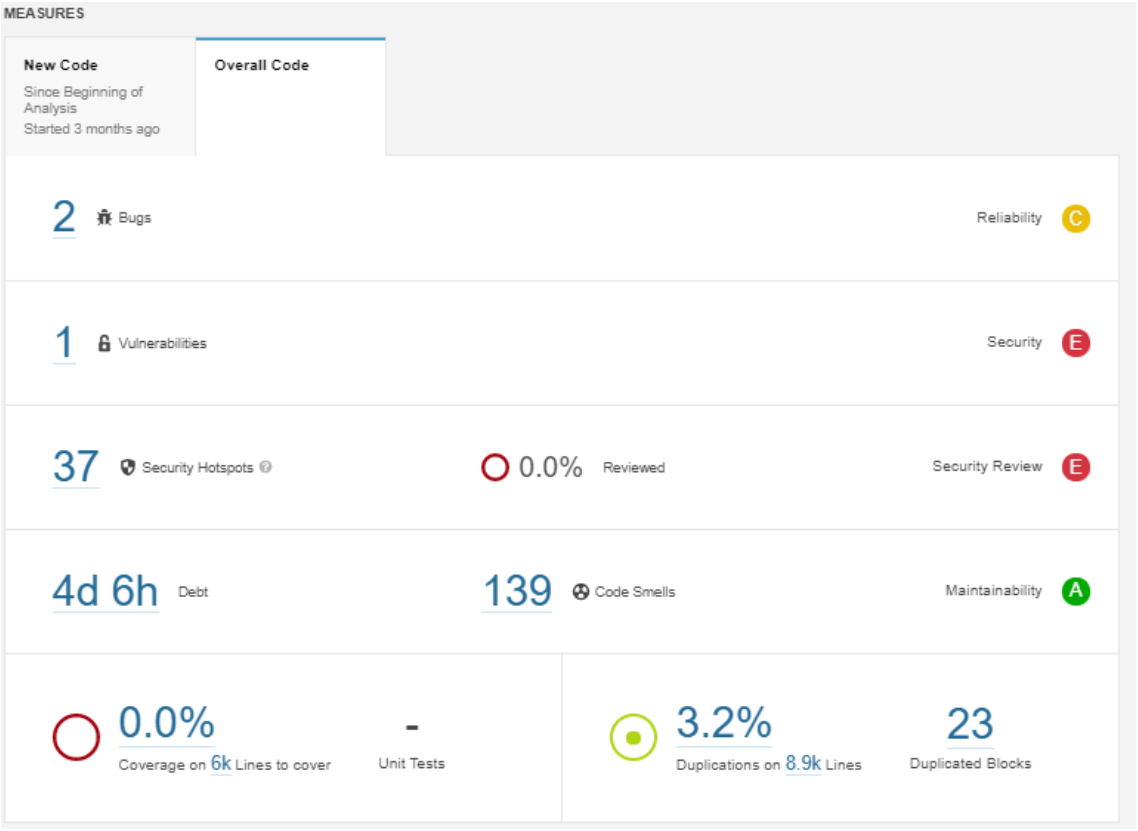


Figura 4.7 – Análise “Final” SonarQube

Conforme a figura 4.7, após as análises e correções finais no código, restaram 2 bugs, 1 vulnerabilidade, 37 security hotspots e 139 code smells, o que indica a correção (ou eliminação) de cerca de 627 code smells, e 102 bugs, ao longo das análises do SonarQube realizadas durante o projeto. O débito restante estimado pelo SonarQube foi 4 dias e 6 horas, que equivale a 38 horas de trabalho, de acordo com as métricas utilizadas pela ferramenta. O débito inicial do projeto era de 14 dias (112 horas, na métrica do SonarQube). Dessa forma, 74 horas foram resolvidos ou excluídos.

Dentre as regras corrigidas, o tempo de manutenção do código inicialmente se manteve bem condizente com a estimativa fornecida pelo SonarQube. Correções repetidas, todavia, foram brevemente mais rápidas de serem corrigidas, visto que o problema já havia sido devidamente interpretado, o que facilitou o processo de manutenção. Apesar disso, as diferenças de tempo em relação à estimativa do SonarQube não passaram de 20%. Após analisar o código, e excluir os falso positivos, o SonarQube demonstrou uma boa estimativa de tempo, em relação aos problemas encontrados no código.

Capítulo 5

Este capítulo tem como objetivo, fornecer ao leitor a descrição da conclusão do projeto. Levando em consideração os resultados obtidos ao longo do projeto.

Conclusão

Garantir uma boa previsão de custo de manutenção é ideal para providenciar dados e informações suficientes para a tomada de decisão da equipe, a respeito das possibilidades do sistema legado no seu respectivo mercado. Nessa etapa, a análise constante do código é realizada, e estabelecer métricas a respeito do esforço a ser adotado, têm um impacto direto na gestão do processo de manutenção do projeto.

As métricas fornecidas pelo SonarQube foram afetadas por conta das repetições das regras infringidas ao longo do código, que acabam por facilitar as respectivas correções após o entendimento do problema encontrado. Além disso, uma grande quantidade de regras foi eliminada inicialmente por apontarem para linhas de código e variáveis geradas automaticamente, na camada View. Por conta disso, antes de considerar devidamente o tempo estimado de manutenção, é importante estudar a estrutura do código, e avaliar as regras infringidas, levando em consideração os tipos, severidades e descrição deles. Esses fatores possibilitaram uma maior celeridade no processo de melhoria/correção do código. Além disso, com a descrição disponibilizada pelo SonarQube a respeito das regras, foi permitido obter um entendimento breve do que ocorre nos trechos de código, assim como possíveis soluções do problema de maneira mais ágil.

No contexto da segurança de dados, a ferramenta também mostrou ser ótima para manifestar possíveis falhas de segurança, principalmente em relação às conexões ao banco de dados.

Apesar da versão Community não conter integração com ferramentas de Integração contínua, como Github, o histórico de versões e atividades, além da possibilidade da atribuição das regras à usuários, permite uma maior organização e controle na qualidade do código.

De maneira geral, por possibilitar a localização de regras infringidas (sejam relacionadas à boas práticas de programação, segurança ou códigos que devem ser corrigidos com urgência), o SonarQube pode ser considerada uma ferramenta de extrema importância, para facilitar o processo de manutenção do código. Ao fornecer um material robusto, referente à descrição das regras infringidas, com exemplos de código e o motivo de serem consideradas problemas, ela também possibilita um bom aprendizado de programação, levantando pontos que, se considerados no processo de desenvolvimento do sistema, geram um código com maior legibilidade e manutenibilidade, que são conceitos primários a serem considerados no desenvolvimento de sistemas.

Capítulo 6

Este capítulo tem como objetivo fornecer ao leitor as referências utilizadas para o desenvolvimento do projeto, assim como fornecer a descrição e possíveis correções dos problemas encontrados pelo SonarQube, no projeto analisado, por meio dos tópicos de Referências Bibliográficas e o apêndice A.

Referências Bibliográficas

LEHMAN, M. M. **Laws of Software Evolution Revisited: 1997**

PRESSMAN, R. S**. Engenharia de Software.** 6. Ed. São Paulo: McGraw-Hill, 2006. 579p.

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson, 2011. 5p. 169p.

SONARQUBE - **Code Quality and Code Security**. Disponível em:

< https://www.sonarqube.org> Último acesso em: 03 mar. 2021.

DEVMEDIA, 2011. **Padrão MVC - Java Magazine**. Disponível em:

<https://www.devmedia.com.br/padrao-mvc-java-magazine/21995#2>.Último acesso em: 16 de Junho de 2021

DEVMEDIA,2008. **Qualidade de Software**.

Disponível em: <https://www.devmedia.com.br/qualidade-de-software/9408>. Último acesso em: 5 de Julho de 2021

DEVMEDIA, 2013. **Princípios da Engenharia de Software**. Disponível em:

<https://www.devmedia.com.br/principios-da-engenharia-de-software/29630#2> Último acesso em: 12 de Junho de 2021.

DEVMEDIA, 2012. **Fundamentos básicos da Orientação a Objetos**. Disponível em: <https://www.devmedia.com.br/fundamentos-basicos-da-orientacao-a-objetos/26372> Último acesso em: 16 de Junho de 2021.

DEVMEDIA, 2010. **Conceitos de Software e Engenharia de Software**.

Disponível em:

<https://www.devmedia.com.br/conceitos-de-software-e-engenharia-de-software/15730> Último acesso em: 16 de Junho de 2021.

Apêndice A

Regras e Melhorias de Código

A análise inicial do projeto possibilitou identificar e separar as regras infringidas por tipo e severidade, além de permitir verificar quais partes do código, incluindo códigos e camadas, foram afetadas.

Code Smells

A análise do código apontou 47 regras infringidas, classificadas como code smells, dentre as severidades Critical, Major, Minor e Info.

Critical

A análise inicial do SonarQube encontrou 8 tipos de regras violadas, de severidade “critical”, contidas na tabela a seguir:

Rule	Quantity	Severity
String literals should not be duplicated	18	Critical
Fields in a "Serializable" class should either be transient or serializable	17	Critical
switch statements should have "default" clauses	4	Critical
Constant names should comply with a naming convention	3	Critical
Instance methods should not write to "static" fields	3	Critical
Methods should not be empty	3	Critical
static base class members should not be accessed via derived types	3	Critical
Cognitive Complexity of methods should not be too high	1	Critical

Tabela 6.1 – Critical Code Smells X Quantidade

String literals should not be duplicated

Camadas afetadas: Model e View

Strings “literais” duplicadas fazem com que o código seja mais propício a erros durante a refatoração, tendo em vista que é necessário atualizar todas as ocorrências. Por outro lado, constantes podem ser referenciadas de vários lugares, mas apenas precisam ser atualizadas em um lugar.

Figura 6.1 – “String literals should not be duplicated” Rule

Fields in a "Serializable" class should either be transient or serializable

Camadas afetadas: View e Utils

Campos em uma classe serializável devem ser serializáveis ou transitórios mesmo que a classe nunca seja explicitamente serializada ou deserializada. Por exemplo, sobrecarga, a maioria dos frameworks de aplicativos J2EE liberam objetos ao disco, e objetos supostamente serializáveis com membros não transientes e não serializáveis podem causar problemas como travamento do programa e abrir a porta para invasões. Em geral, espera-se que uma classe serializável cumpra seu papel e não tenha um comportamento inesperado quando uma instância é serializada.

Figura 6.2 – “Fields in a ‘Serializable’ class should either be transient or serializable Rule”

Switch statements should have "default" clauses

Camadas afetadas: Control e View

O requisito para uma cláusula final “default” é a programação defensiva. A cláusula deve ou tomar uma ação apropriada, ou conter um comentário a respeito do motivo de nenhuma ação ter sido executada.

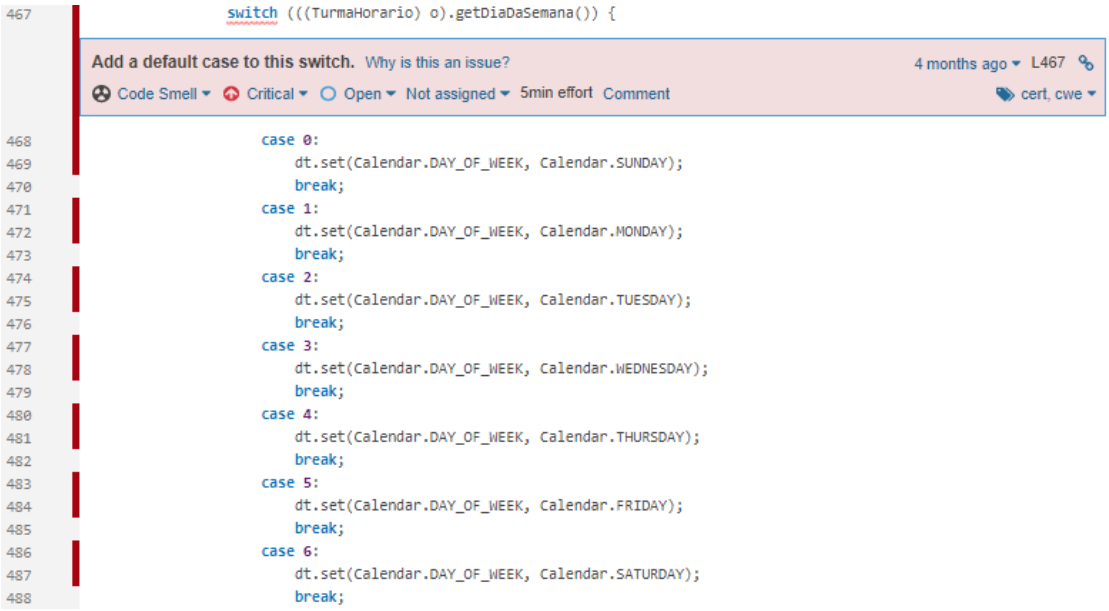


Figura 6.3 – “’Switch statements should have ‘default’ clauses” Rule

Constant names should comply with a naming convention

Camadas afetadas: Model

Convenções compartilhadas de código permitem que as equipes colaborem eficientemente. Essa regra verifica se o nome de todas constantes correspondem à uma expressão regular disponível.

Figura 6.4 – “Constant names should comply with a naming Convention” Rule

Instance methods should not write to "static" fields

Camada afetada: Model

Atualizar corretamente campos estáticos de métodos não estáticos é trabalhoso e pode facilmente causar bugs se houver múltiplas instâncias de classe e/ou múltiplas threads. Idealmente, campos estáticos só são atualizados a partir de métodos estáticos sincronizados.

Figura 6.5 – “Instance methods should not write to ‘static’ fields” Rule

Methods should not be empty

Camadas afetadas: View

Há vários motivos para um método não ter um corpo:

- É uma omissão não intencional, e deve ser arrumada para prevenir comportamentos não esperados no ambiente de produção.
- Não é, ou nunca vai ser suportado. Nesse caso, uma exceção “UnsupportedOperationException” should be thrown.
- O método é uma sobrescrição intencionalmente vazia. Nesse caso, um comentário deve explicar o motivo do uso da sobrescrição em branco.

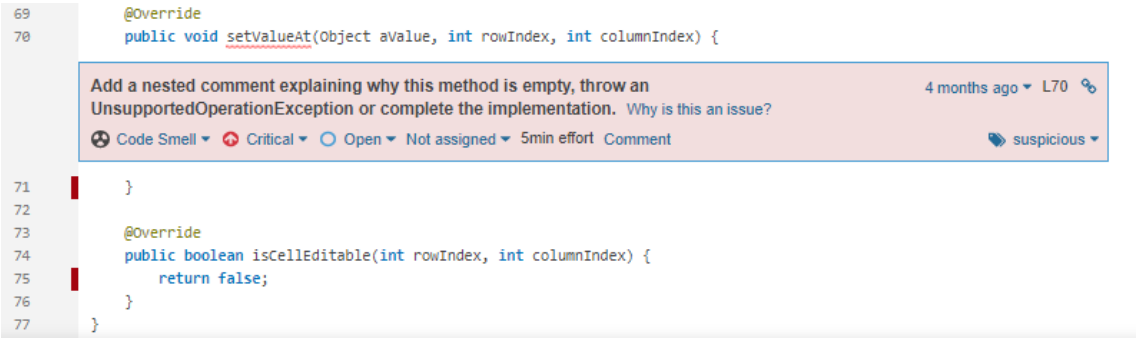


Figura 6.6 - “Methods should not be empty” Rule

Static base class members should not be accessed via derived types

Camada afetada: View

No interesse de um código claro, membros estáticos de uma classe base nunca devem ser acessados utilizando tipos derivados de nomes. Fazer isso é confuso, cria a ilusão que dois membros estáticos diferentes existem.

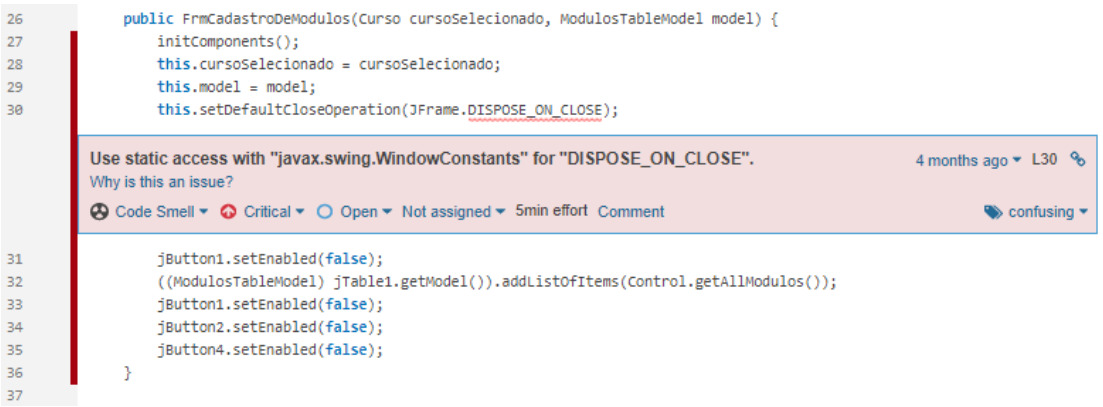
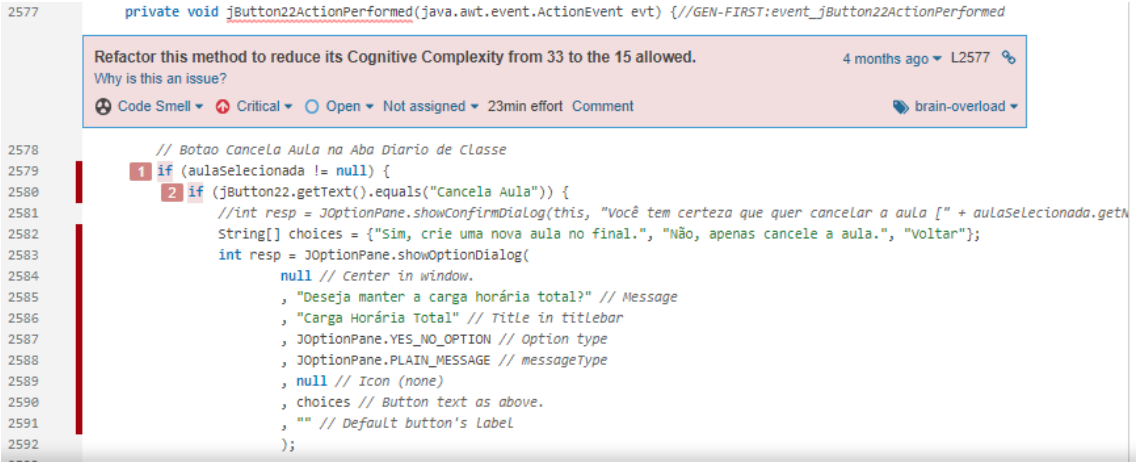


Figura 6.7 – “’static’ base class members should not be accessed via derived types” Rule

Cognitive Complexity of methods should not be too high

Camada afetada: View

Complexidade cognitiva é uma medida do quão difícil o fluxo de controle de um método é, de ser entendido. Métodos com alta complexidade cognitiva são difíceis de serem mantidos.




```
2577    private void jButton22ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-FIRST:event_jButton22ActionPerformed
2578        // Botao Cancela Aula na Aba Diario de Classe
2579        if (aulaSelecionada != null) {
2580            if (jButton22.getText().equals("Cancela Aula")) {
2581                //int resp = JOptionPane.showConfirmDialog(this, "Você tem certeza que quer cancelar a aula [" + aulaSelecionada.getN
2582                String[] choices = {"Sim, crie uma nova aula no final.", "Não, apenas cancele a aula.", "Voltar"};
2583                int resp = JOptionPane.showOptionDialog(
2584                        null // Center in window.
2585                        , "Deseja manter a carga horária total?" // Message
2586                        , "Carga Horária Total" // Title in titlebar
2587                        , JOptionPane.YES_NO_OPTION // Option type
2588                        , JOptionPane.PLAIN_MESSAGE // messageType
2589                        , null // Icon (none)
2590                        , choices // Button text as above.
2591                        , "" // Default button's label
2592                        );
```

Figura 6.8 – “Cognitive Complexity of methods should not be too high” Rule

Major

A análise inicial do SonarQube encontrou 19 tipos de regras violadas, de severidade “major”, contidas na tabela a seguir:

Rule	Quantity	Severity
Unused method parameters should be removed	95	Major
Raw types should not be used	82	Major
Anonymous inner classes containing only one method should become lambdas	76	Major
@Override should be used on overriding and implementing methods	29	Major
Sections of code should not be commented out	25	Major
Standard outputs should not be used directly to log anything	19	Major
Source files should not have any duplicated blocks	11	Major
Inheritance tree of classes should not be too deep	6	Major
Constructors should not be used to instantiate "String", "BigInteger", "BigDecimal" and primitive-wrapper classes	4	Major
Methods should not have too many parameters	3	Major
Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used	3	Major
A field should not duplicate the name of its containing class	2	Major
Collapsible "if" statements should be merged	2	Major
Nested blocks of code should not be left empty	2	Major
Constructors of an "abstract" class should not be declared "public"	1	Major
Generic exceptions should never be thrown	1	Major
Local variables should not shadow class fields	1	Major
Unused assignments should be removed	1	Major
Utility classes should not have public constructors	1	Major

Tabela 6.2 – Major Code Smells x Quantidade

Unused method parameters should be removed

Camada afetada: View

Parâmetros não utilizados podem induzir erros. Não importa o valor atribuído a tais parâmetros, o comportamento será o mesmo.

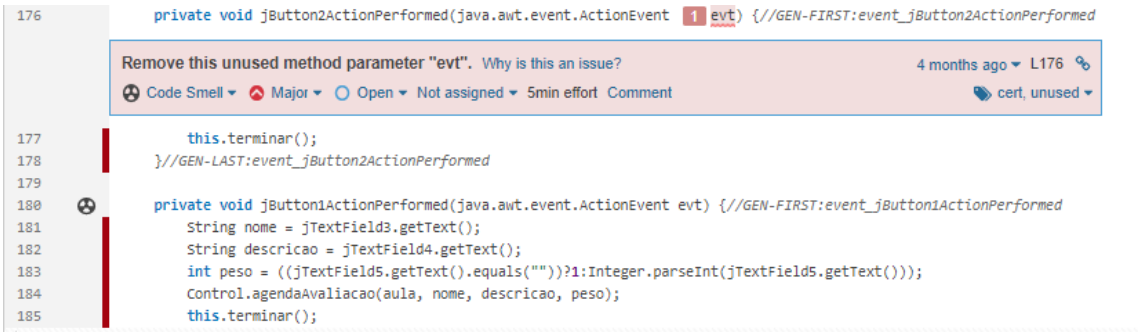


Figura 6.9 – “Unused method parameters should be removed” Rule

Nesses casos, os parâmetros evt foram criados automaticamente, na criação dos eventos de clique na interface do sistema.

Por conta disso, tais regras foram ignoradas na análise. Para manter a rastreabilidade das mudanças, foi criado um evento no gráfico de acompanhamento de atividade, do próprio SonarQube.

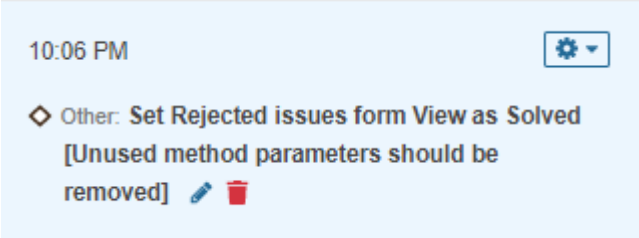


Figura 6.10 – Activity Story: “Unused method parameters should be removed” cases solved

Raw types should not be used

Camadas afetadas: View e Utils

Tipos genéricos não devem ser utilizados brutos (sem parâmetros de tipo) em declarações de variáveis ou valores de retorno. Isso ignora a verificação de tipo genérico e adia a captura de código inseguro durante o tempo de execução do código.

Figura 6.11 – “Raw types should not be used” Rule

O sonarqube apontou como violações de regras, códigos gerados automaticamente na criação de elementos da interface do sistema. Tais regras foram ignoradas.

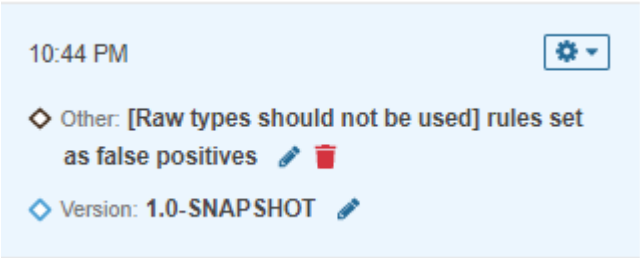


Figura 6.12 - Activity Story: “Raw types should not be used” cases solved

Anonymous inner classes containing only one method should become lambdas

Camada afetada: View

Antes do Java 8, a única maneira de providenciar suporte parcial a encerramentos em Java era por meio do uso de classes internas anônimas. Porém, a sintaxe das classes anônimas é complicada e confusa. Com o Java 8, a maioria dos usos de classes internas anônimas deve ser substituída por lambdas para aumentar bastante a legibilidade do código-fonte.

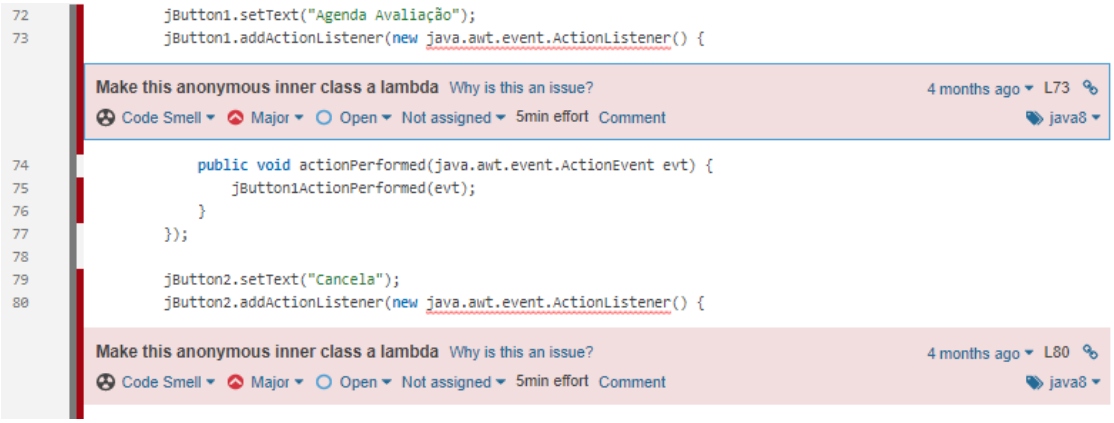


Figura 6.13 – “Anonymous inner classes containing only one method should become lambdas” Rule

Novamente, as regras apontadas pelo sonarqube pertencem à camada View, e aponta códigos gerados automaticamente, relacionados à interface do sistema. Dessa forma, essa regra foi ignorada.

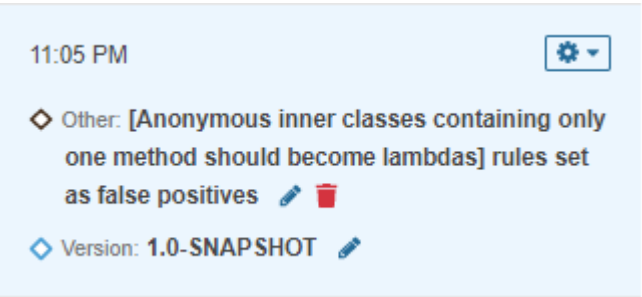


Figura 6.14 - Activity Story: “Anonymous inner classes containing only one method should become lambdas” cases solved

@Override should be used on overriding and implementing methods

Camadas afetadas: Model e View

Utilizar a anotação @override têm 2 motivos úteis:

- Ele provoca um aviso ao compilador, caso o método não sobrescreva nada, como no caso de erros de código.
- Ele melhora a legibilidade do código fonte, por tornar óbvio quais métodos são sobrescritos.

Figura 6.15 - "'@Override' should be used on overriding and implementing methods" Rule

Código compatível:

Figura 6.16 - "'@Override' should be used on overriding and implementing methods" Compatible Code

As demais regras infligidas na camada View, implicam em códigos gerados automaticamente e não editáveis, nos códigos fonte dos formulários, portanto, foram eliminados da análise.

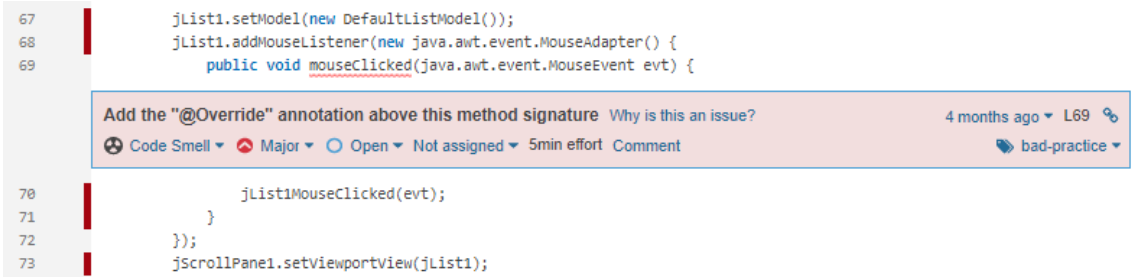


Figura 6.17 – "@Override' should be used on overriding and implementing methods" Camada View

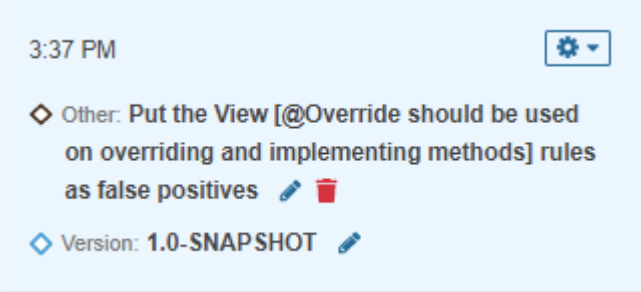


Figura 6.18 - Activity Story: "@Override' should be used on overriding and implementing methods" cases solved

Sections of code should not be commented out

Camadas afetadas: Control, Model e View

Programadores não devem comentar códigos, visto que aumentam o código (Bloat) e reduzem a legibilidade. Códigos não utilizados devem ser excluídos e recuperados no histórico de controle do código, caso necessário.

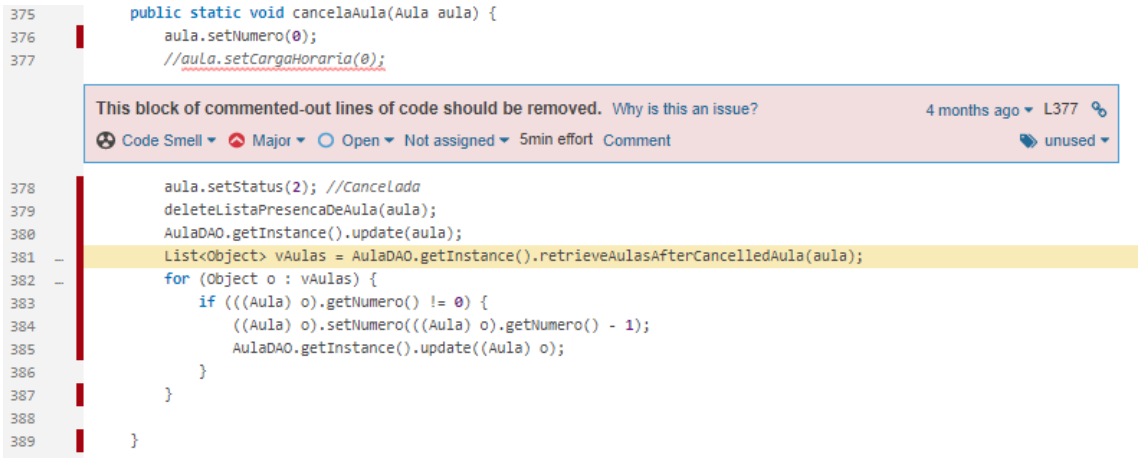


Figura 6.19 – "Sections of code should not be commented out" Rule

Código compatível:

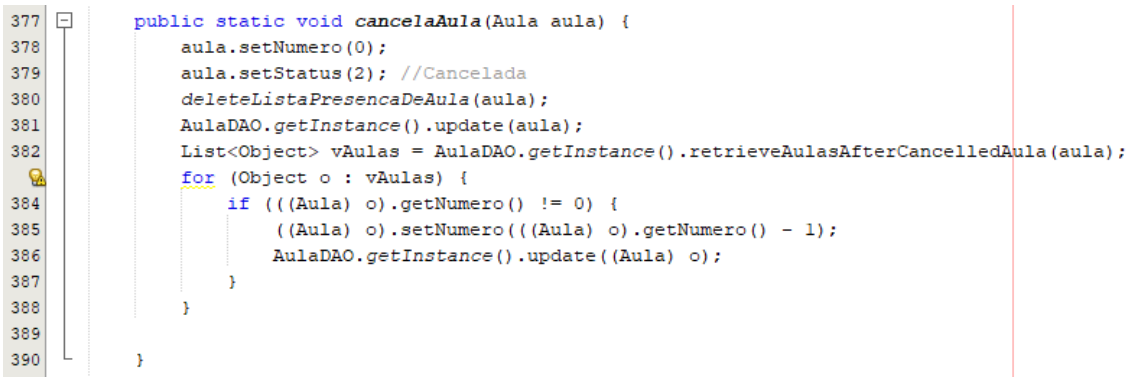


Figura 6.20 – "Sections of code should not be commented out" Código Compatível

Standard outputs should not be used directly to log anything

Camadas afetadas: Control, Model e View

Ao registrar uma mensagem do sistema, há vários requisitos importantes que devem ser satisfeitos:

- O usuário deve ser capaz de obter os logs facilmente
- O formato das mensagens registradas deve ser uniformes e possibilitar ao usuário, fácil leitura do log
- Os dados registrados devem ser armazenados
- Dados sensíveis devem ser registrados de maneira segura

Se um programa escreve diretamente os outputs, não há como satisfazer tais requisitos citados. Por conta disso, recomenda-se a definição e uso de um logger dedicado.

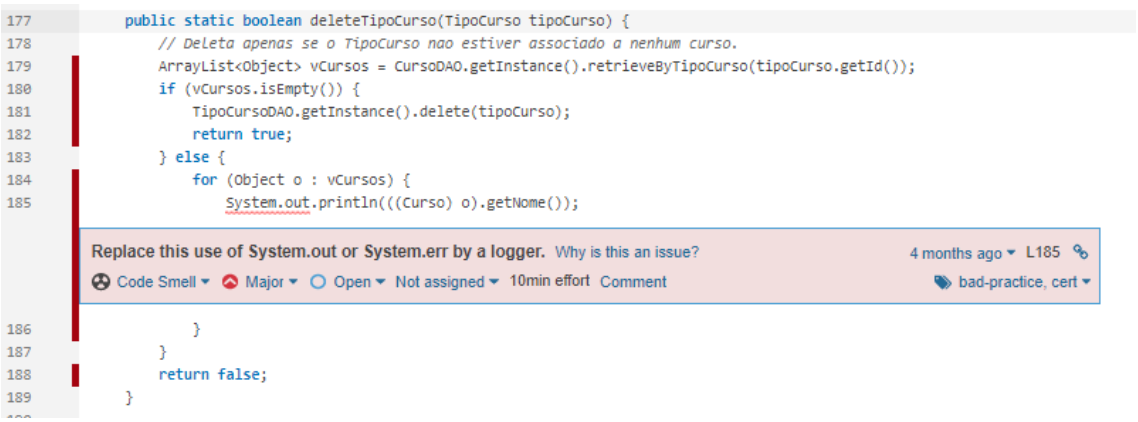


Figura 6.21 – “Standard outputs should not be used directly to log anything” Rule

Source files should not have any duplicated blocks

Camadas afetadas: Model e View

Um problema é criado em um arquivo, assim que há ao menos um bloco duplicado no arquivo.

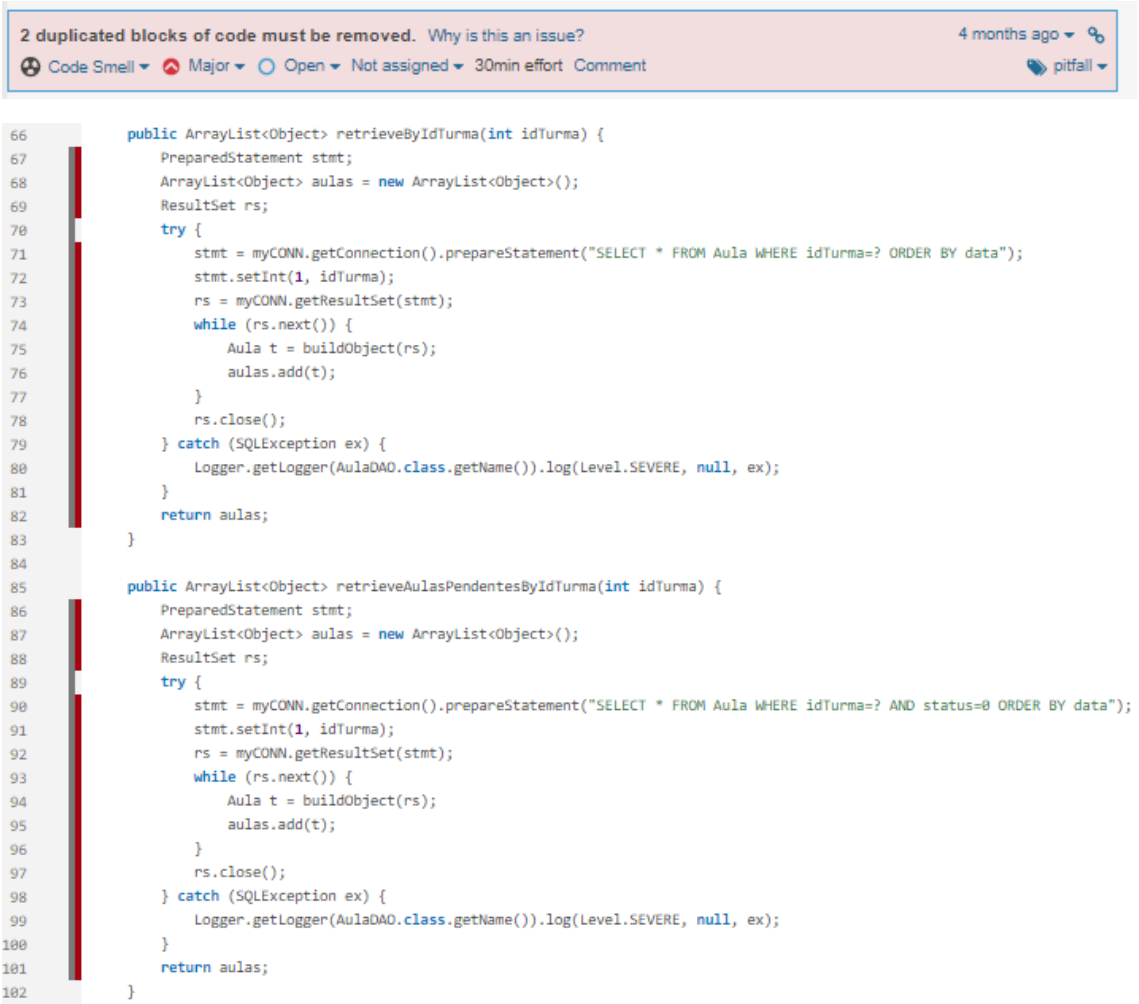


Figura 6.22 – “Source files should not have any duplicated blocks” Rule

Inheritance tree of classes should not be too deep

Camada afetada: View

As heranças são com certeza, um dos conceitos mais importantes da programação orientada a objetos. É um jeito de compartimentar e reutilizar códigos, por meio da criação de coleções de atributos e comportamentos chamados de classes, que podem ser baseados em classes criadas anteriormente. Porém, o abuso desse conceito, ao criar uma árvore de herança complexa pode aumentar muito a complexidade de um código, tornando cada vez mais difícil a manutenibilidade do código. Na maioria das vezes, uma árvore de herança muito profunda está relacionada a um design orientado a objeto falho, que gerou o uso sistemático de heranças, enquanto o uso da composição seria melhor.

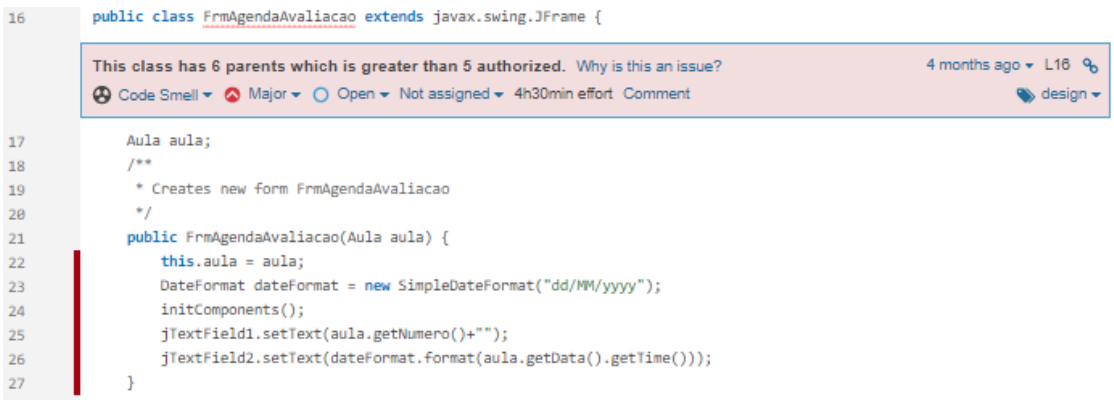


Figura 6.23 – “Inheritance tree of classes should not be too deep” Rule

A única camada afetada foi a View, sendo também, relacionada à

Constructors should not be used to instantiate "String", "BigInteger", "BigDecimal" and primitive-wrapper classes

Camadas afetadas: Model e View

Construtores para String, BigInteger, BigDecimal e objetos utilizados para envolver variáveis primitivas não devem ser utilizados. Realizar isso é menos claro e utiliza mais memória do que simplesmente utilizar o valor desejado no caso de Strings, e usar ValueOf para o restante.

Figura 6.24 – “Constructors should not be used to instantiate ‘String’, ‘BigInteger’, ‘BigDecimal’ and primitive-wrapper classes” Rule

Methods should not have too many parameters

Camadas afetadas: Model

Uma lista de parâmetros longa pode indicar que uma nova estrutura deveria ser criada para envolver inúmeros parâmetros ou que a função está fazendo muitas coisas.

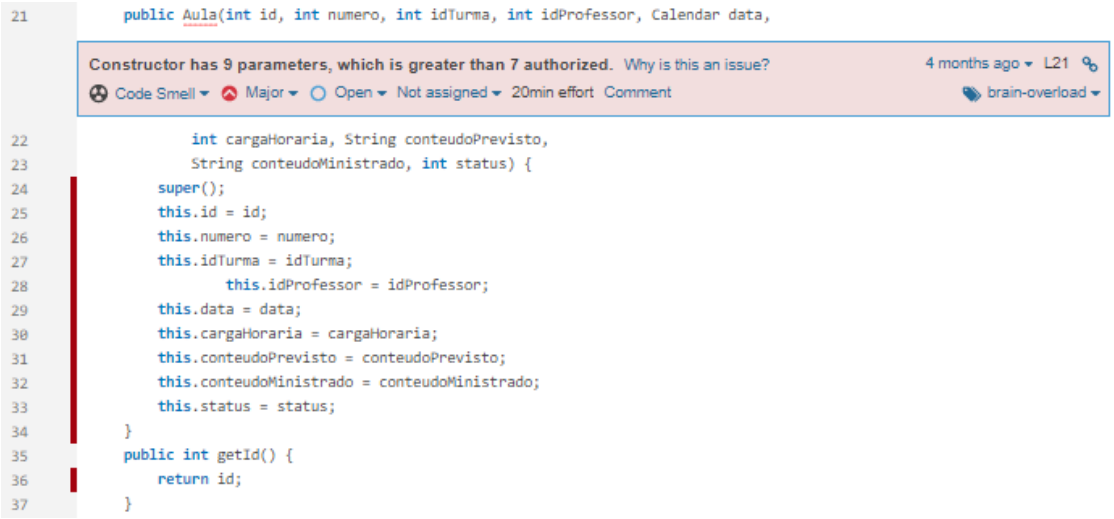


Figura 6.25 – “Methods should not have too many parameters” Rule

Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used

Camada afetada: View

As primeiras classes do JAVA API, como Vector, Hashtable e StringBuffer, eram sincronizados para que fossem “thread-safe”. Infelizmente, a sincronização possui um grande impacto negativo de performance, mesmo quando tais coleções são utilizadas por apenas um thread.

Nesse caso, é melhor utilizar seus novos substitutos não sincronizados:

- ArrayList ou LinkedList, ao invés de Vector
- Deque, ao invés de Stack
- HashMap, ao invés de Hashtable
- StringBuilder, ao invés de StringBuffer

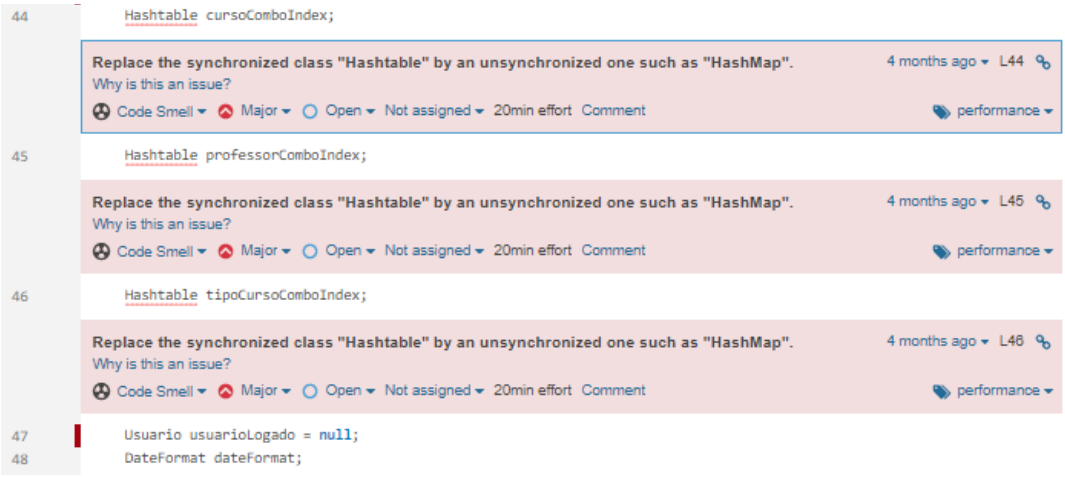


Figura 6.26 – “Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used” Rule

A field should not duplicate the name of its containing class

Camada afetada: Model

É confuso ter um membro de classe com o mesmo nome como sua classe envolvente. Isso é especialmente verdade quando você considera a prática comum e nomear uma instância da classe para a própria classe.

A boa prática determina que qualquer campo ou membro com o mesmo nome que a classe envolvente seja nomeado de forma a ser mais descritivo a respeito do aspecto que a classe representa ou mantém.

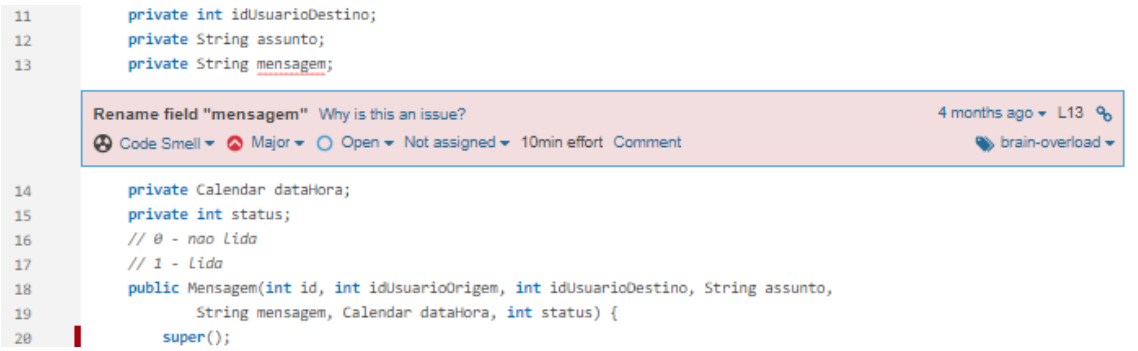


Figura 6.27 – “A field should not duplicate the name of its containing class” Rule

Collapsible "if" statements should be merged

Camada afetada: View

Unificar as instruções if’s aumenta a legibilidade do código.

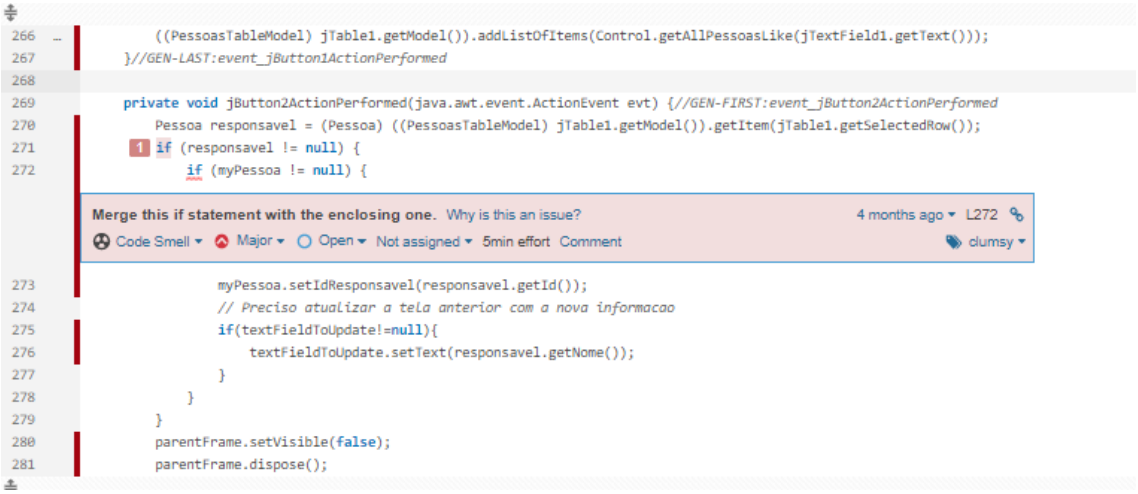


Figura 6.28 – “Collapsible ‘if’ statements should be merged” Rule

Nested blocks of code should not be left empty

Camada afetada: Model

Na maioria das vezes que um bloco de código se encontra vazio por conta de um pedaço do código se encontra ausente. Nesses casos, é necessário que o bloco seja removido ou preenchido.

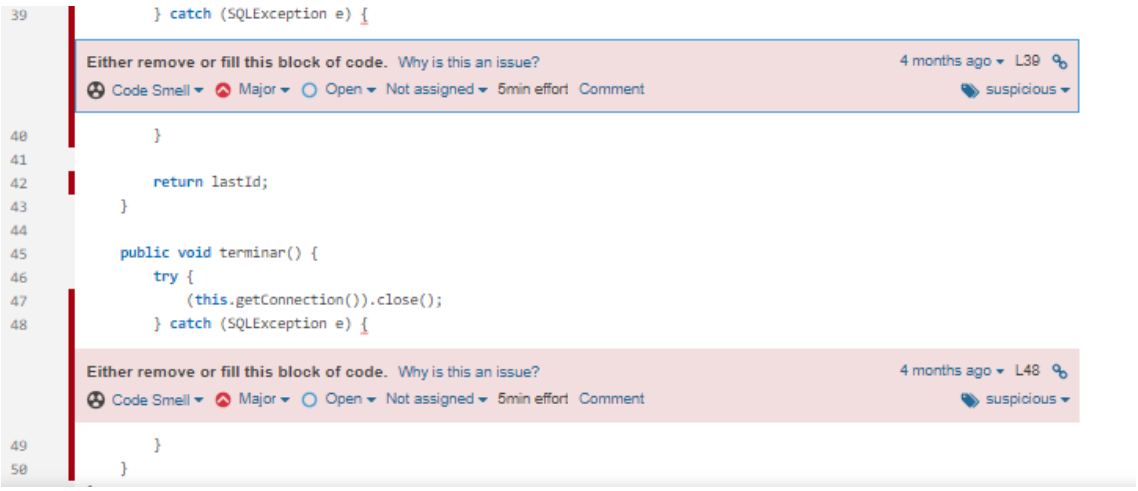


Figura 6.29 - Nested blocks of code should not be left empty

Constructors of an "abstract" class should not be declared "public"

Camada afetada: Utils

Classes abstratas não devem ter construtores públicos. Construtores de classes abstratas só podem ser instanciadas em construtores de suas sub-classes. Por conta disso, não há necessidade de manter eles públicos. O modificador Protected deve ser o suficiente.

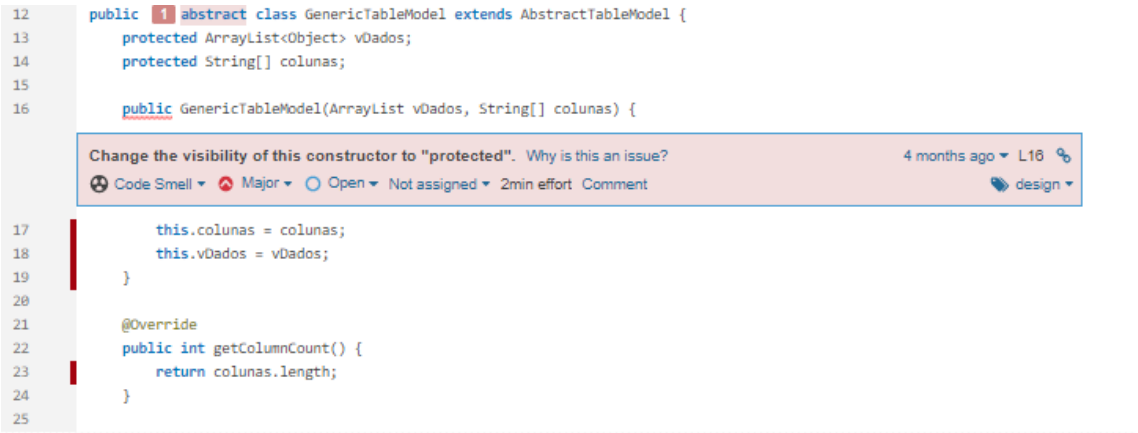


Figura 6.30 – Constructors of an "abstract" class should not be declared "public"

Generic exceptions should never be thrown

Camada afetada: Model

Utilizar exceções genéricas como Error, RuntimeException, Throwable e Exception evita que os métodos chamados tratem exceções “true” geradas pelo sistema de maneiras diferentes dos erros gerados pelo aplicativo.

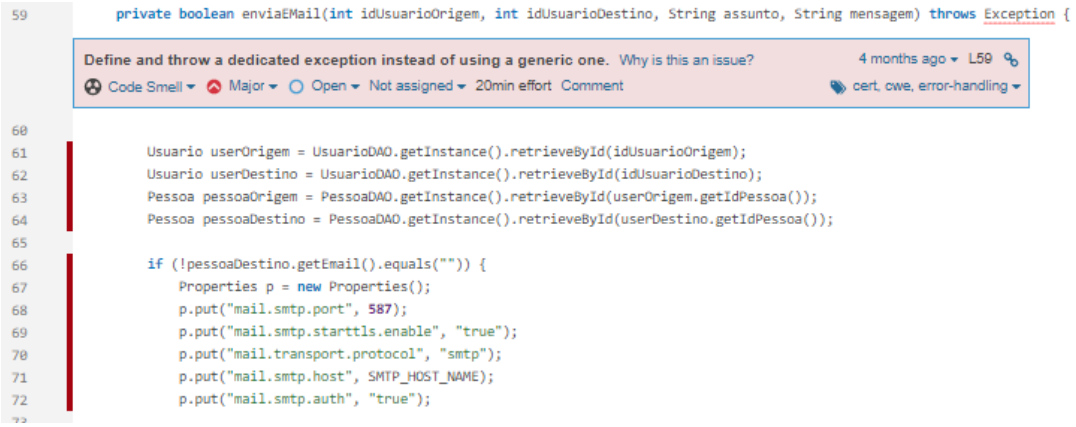


Figura 6.31 - Generic exceptions should never be thrown

Local variables should not shadow class fields

Camada afetada: Model

Sobrescrever ou obscurecer uma variável declarada pode impactar a legibilidade e manutenibilidade de um trecho de código. Além disso, também pode induzir os programadores a introduzir bugs, visto que pensam estar utilizando uma variável, enquanto estão, na verdade, utilizando outra.

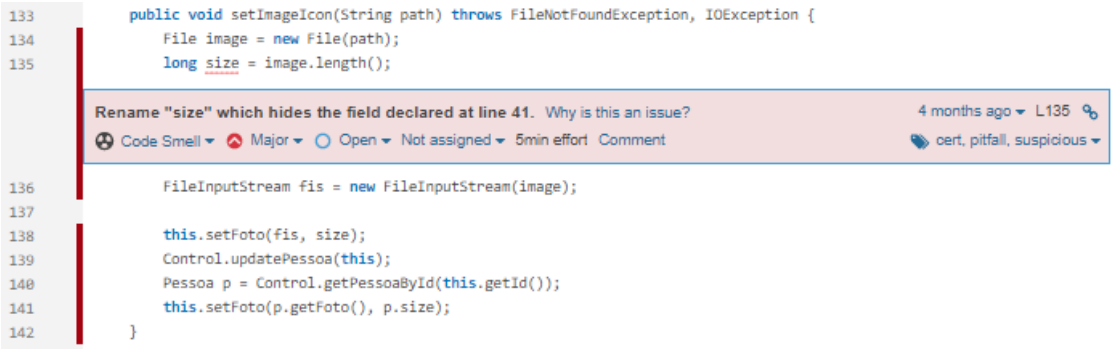


Figura 6.32 - Local variables should not shadow class fields

Unused assignments should be removed

Camada afetada: View

Um armazenamento “morto” ocorre quando uma variável local é atribuída a um valor que não é lido por nenhuma instrução subsequente. Calcular ou recuperar um valor apenas para substituir ou descartar o mesmo pode indicar um erro grave de código. Mesmo que não seja um erro, na melhor das hipóteses, é um desperdício de recursos e portanto, todos valores calculados devem ser utilizados.

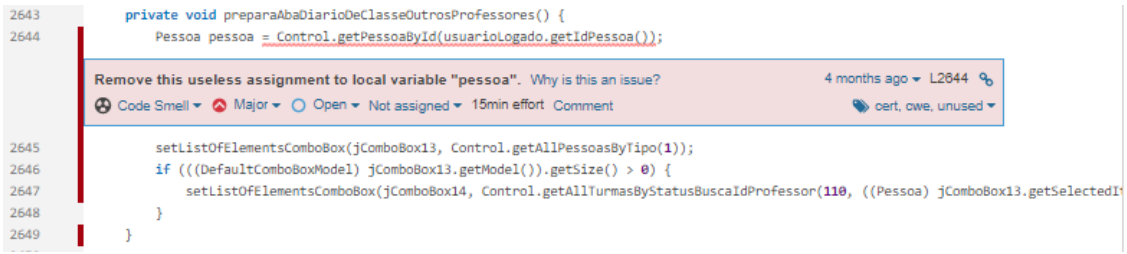


Figura 4.33 - Unused assignments should be removed

Utility classes should not have public constructors

Camada afetada: Control

Classes utilitárias, que são coleções de membros estáticos, não devem ser instanciadas. Mesmo as classes utilitárias abstratas, que podem ser extendidas, não devem ter construtores públicos. O Java adiciona um construtor público implícito para cada classe que não define ao menos um explicitamente. Portanto, ao menos um construtor não público deve ser definido.

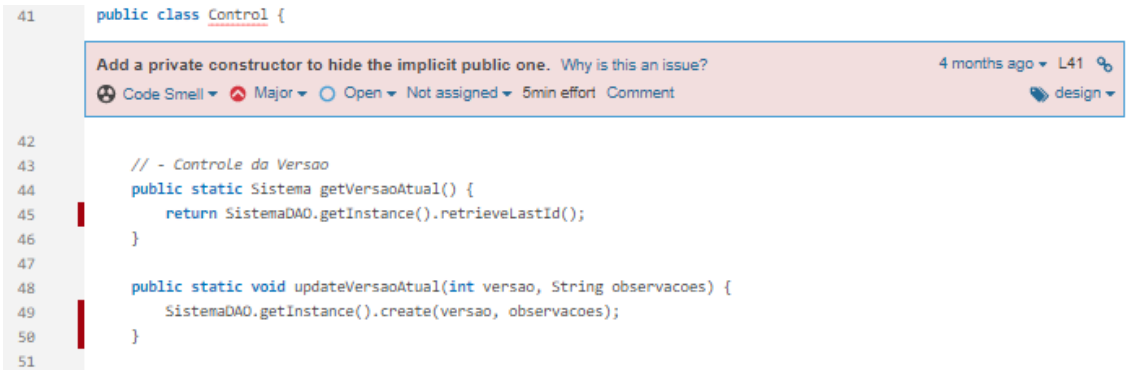


Figura 6.34 - Utility classes should not have public constructors

Minor

A análise inicial do SonarQube encontrou 19 tipos de regras violadas, de severidade “minor”, contidas na tabela a seguir:

Rule	Quantity	Severity
Private fields only used as local variables in methods should become local variables	152	Minor
Declarations should use Java collection interfaces such as "List" rather than specific implementation classes such as "LinkedList"	76	Minor
The diamond operator ("<>") should be used	40	Minor
Catches should be combined	18	Minor
@Deprecated code should not be used	16	Minor
Unnecessary imports should be removed	9	Minor
Array designators "[]" should be on the type, not the variable	6	Minor
Boolean literals should not be redundant	6	Minor
Return of boolean expressions should not be wrapped into an "if-then-else" statement	6	Minor
Redundant casts should not be used	5	Minor
Boxed "Boolean" should be avoided in boolean expressions	3	Minor
Multiple variables should not be declared on the same line	2	Minor
Public constants and fields initialized at declaration should be "static final" rather than merely "final"	2	Minor
Unused local variables should be removed	2	Minor
Class variable fields should not have public accessibility	1	Minor
Local variables should not be declared and then immediately returned or thrown	1	Minor
Static non-final field names should comply with a naming convention	1	Minor
switch statements should have at least 3 "case" clauses	1	Minor
throws declarations should not be superfluous	1	Minor

Tabela 6.3 - Minor Code Smells x Quantidade

<u>Private fields only used as local variables in methods should become local variables</u>

Camadas afetadas: Model e View

Quando o valor de um campo privado é sempre aderido aos métodos de uma classe, antes de ser lido, então ele não está sendo utilizado para armazenar informações de classe. Dessa forma, ele deve se tornar uma variável local nos respectivos métodos relevantes, para evitar enganos.

Figura 6.35 – Private fields only used as local variables in methods should become local variables

Código compatível:

```
143     public ImageIcon getImageIcon() throws IOException {
144         ImageIcon myImageIcon;
145         Image img = ImageIO.read(new BufferedInputStream(this.getFoto()));
146 
147         ImageIcon icon = new ImageIcon(img);
148         Image newImg = icon.getImage();
149         Image resizedImg = newImg.getScaledInstance(170, 230, Image.SCALE_SMOOTH);
150 
151         myImageIcon = new ImageIcon(resizedImg);
152         return myImageIcon;
153     }
154 
155 }
```

Figura 6.36 - Private fields only used as local variables in methods should become local variables Código Compatível

OBS: Sonarqube acaba por considerar a declaração automática das variáveis dos botões do painel Java, como violações de regras. Nesses casos, tais declarações não foram consideradas violações, e não foram “corrigidas”.

```
222     // Variables declaration - do not modify
223     private javax.swing.JButton jButton1;
        private javax.swing.JButton jButton2;
225     private javax.swing.JLabel jLabel1;
226     private javax.swing.JLabel jLabel2;
227     private javax.swing.JLabel jLabel3;
228     private javax.swing.JLabel jLabel4;
229     private javax.swing.JLabel jLabel5;
230     private javax.swing.JPanel jPanel1;
231     private javax.swing.JTextField jTextField1;
232     private javax.swing.JTextField jTextField2;
233     private javax.swing.JTextField jTextField3;
234     private javax.swing.JTextField jTextField4;
235     private javax.swing.JTextField jTextField5;
236     // End of variables declaration
237 }
```

Figura 6.37 – Private fields only used as local variables in methods should become local variables Código Compatível - View

Declarations should use Java collection interfaces such as "List" rather than specific implementation classes such as "LinkedList"

Camadas afetadas: Control, Model, Utils e View

O propósito da API “Java Collections” é prover uma hierarquia de interfaces bem definida, para mascarar os detalhes de implementação.

A implementação de classes deve ser utilizada apenas para instanciar novas coleções. Dessa forma, o resultados de uma instância devem ser armazenadas em uma variável pertencente ao Java Collection.

Essa regra é violada quando a classe implementada é:

- Retornada por um método público
- Aceita como um argumento, por um método público
- Exposta por um membro público

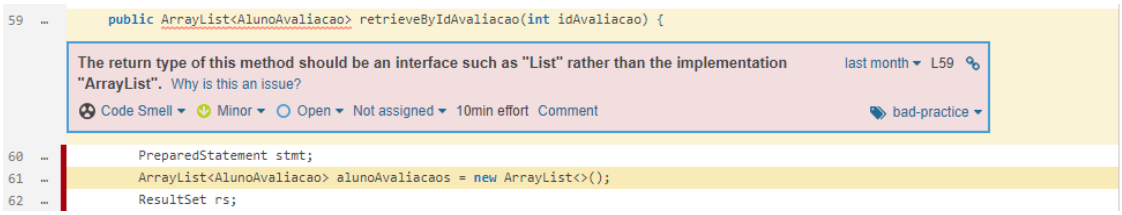


Figura 6.38 - Declarations should use Java collection interfaces such as "List" rather than specific implementation classes such as "LinkedList"

Código compatível:

```
public List<AlunoAvaliacao> retrieveByIdAvaliacao(int idAvaliacao) {
    PreparedStatement stmt;
    List<AlunoAvaliacao> alunoAvaliacaos = new ArrayList<>();
    ResultSet rs;
    try {
```

```
private static Aula createNextAula(Aula aula) {
    List<Object> vTurmaHorario = getAllTurmaHorarioByIdTurma(aula.getIdTurma());
    Calendar dt = Calendar.getInstance();
    dt.setTime(aula.getData().getTime());
    dt.add(Calendar.DAY_OF_YEAR, 1);
    while (!isDateInTurmaHorario(dt, (ArrayList<Object>) vTurmaHorario)) {
        dt.add(Calendar.DAY_OF_YEAR, 1);
    }
    Aula novaAula = createAula(aula.getNumero() + 1, aula.getIdTurma(), dt, aula.getCargaHoraria());
    createListaPresencaDeAula(novaAula);
    return novaAula;
}
```

Figura 6.39 - Declarations should use Java collection interfaces such as "List" rather than specific implementation classes such as "LinkedList" Código Compatível

The diamond operator ("<>") should be used

Camadas afetadas: Model e View

Para reduzir o detalhamento do código, o Java 7 introduziu o operador “diamante”. Com ele, não é necessário declarar um tipo de lista na declaração e no construtor, levando em consideração que com o operador diamante, é possível simplificar a declaração do construtor para <>, enquanto o compilador irá inferir o respectivo tipo.

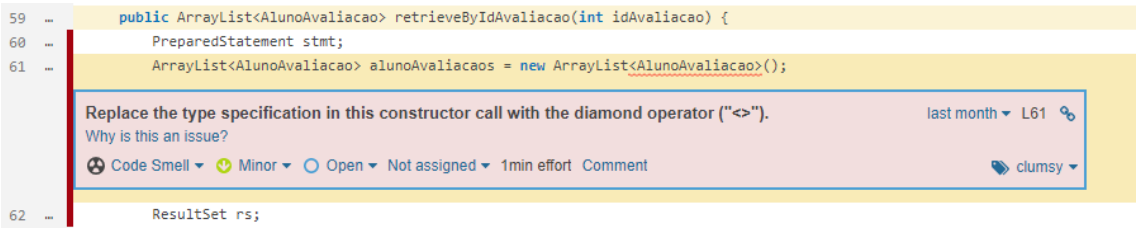


Figura 6.40 - The diamond operator ("<>") should be used

Código Compatível:

```
59  public ArrayList<AlunoAvaliacao> retrieveByIdAvaliacao(int idAvaliacao) {
60      PreparedStatement stmt;
        ArrayList<AlunoAvaliacao> alunoAvaliacaos = new ArrayList<>();
62      ResultSet rs;
63      try {
64          stmt = myCONN.getConnection().prepareStatement("SELECT * FROM AlunoAvaliacao WHERE idAvaliacao=?");
65          stmt.setInt(1, idAvaliacao);
66          rs = myCONN.getResultSet(stmt);
67          while (rs.next()) {
68              AlunoAvaliacao t = buildObject(rs);
69              alunoAvaliacaos.add(t);
70          }
71          rs.close();
72      } catch (SQLException ex) {
73          Logger.getLogger(AlunoAvaliacaoDAO.class.getName()).log(Level.SEVERE, null, ex);
74      }
75      return alunoAvaliacaos;
76  }
```

Figura 6.41 – The diamond operator ("<>") should be used Código Compatível

Catches should be combined

Camada afetada: View

A partir do Java 7, é possível “apanhar” múltiplas exceções de uma só vez. Dessa forma, quando blocos “Catch” possuem o mesmo código, eles devem ser combinados, para manter uma boa legibilidade do código.

Figura 6.42 - Catches should be combined

Código compatível:

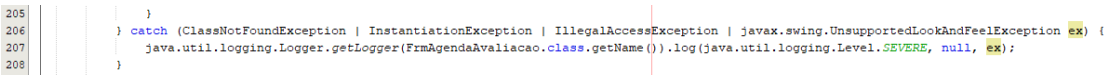


Figura 6.43 – Catches should be combined Código Compatível

@Deprecated code should not be used

Camada afetada: Model

O SonarQube descreve que classes e interfaces em descontinuação, não devem ser utilizados no código.

Conforme a análise do código, o método newInstance() deve ser substituído.

Figura 6.44 - @Deprecated code should not be used

Código compatível:

```
private AlunoAvaliacaoDAO() {
    try {
        Class<?> daoClass = Class.forName("model." + GenericCONN.DAOName);
        myCONN = (GenericCONN) daoClass.getDeclaredConstructor().newInstance();
    } catch (Exception e) {
        System.out.println("Could not find DAO class: " + "model." + GenericCONN.DAOName);
    }
    myCONN.getConnection();
}
```

Figura 6.45 - @Deprecated code should not be used Código Compatível

Unnecessary imports should be removed

Camadas afetadas: Model e View

As importações utilizadas no código, devem ser criadas pelo próprio IDE, e não manualmente, pelo desenvolvedor. Importações não utilizadas ou desnecessárias não devem ocorrer, visto que podem afetar negativamente a legibilidade do código.

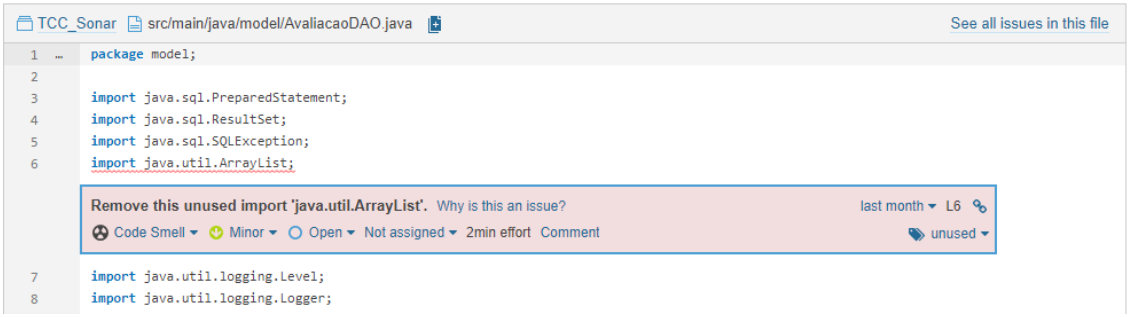


Figura 6.46 – Unnecessary imports should be removed

Código compatível:

```
1  package model;
2
3  import java.sql.PreparedStatement;
4  import java.sql.ResultSet;
5  import java.sql.SQLException;
6  import java.util.logging.Level;
7  import java.util.logging.Logger;
8
```

Figura 6.47 – Unnecessary imports should be removed Código Compatível

Array designators "[]" should be on the type, not the variable

Camada afetada: View

Os designadores array devem sempre estar localizados no tipo, para manter a legibilidade do código. Caso contrário, os desenvolvedores precisam verificar ambos os tipos e o nome da variável para entender se uma variável é ou não, um array.

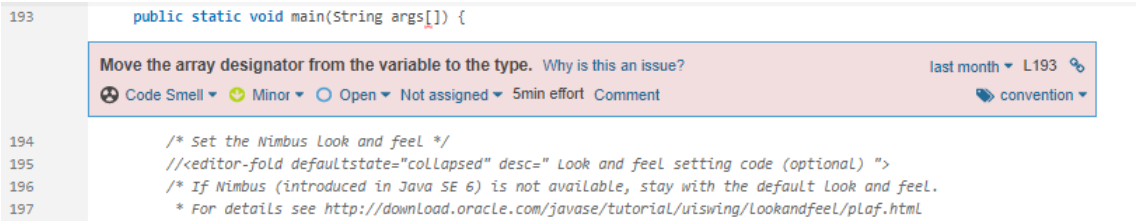


Figura 6.48 - Array designators "[]" should be on the type, not the variable

Código compatível

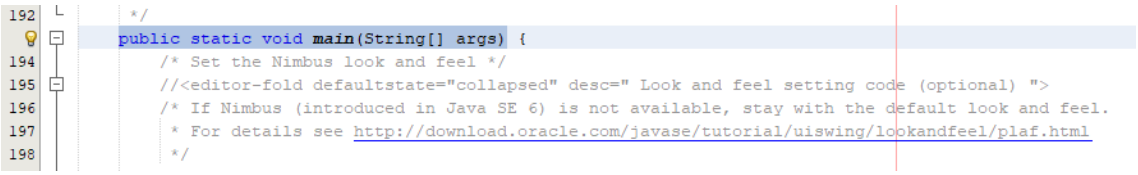


Figura 6.49 – Array designators "[]" should be on the type, not the variable
Código Compatível

Boolean literals should not be redundant

Camadas afetadas: Model e View

Literais booleanas redundantes devem ser removidos das expressões, para melhorar a legibilidade.

Figura 6.50 - Boolean literals should not be redundant

Return of boolean expressions should not be wrapped into an "if-then-else" statement

Camada afetada: View

O retorno gerado por instruções literais booleanas agrupadas em instruções if-then-else deve ser simplificado. Da mesma forma, as invocações de métodos agrupados em if-then-else diferindo apenas de literais booleanos devem ser simplificados por meio de uma única instância.

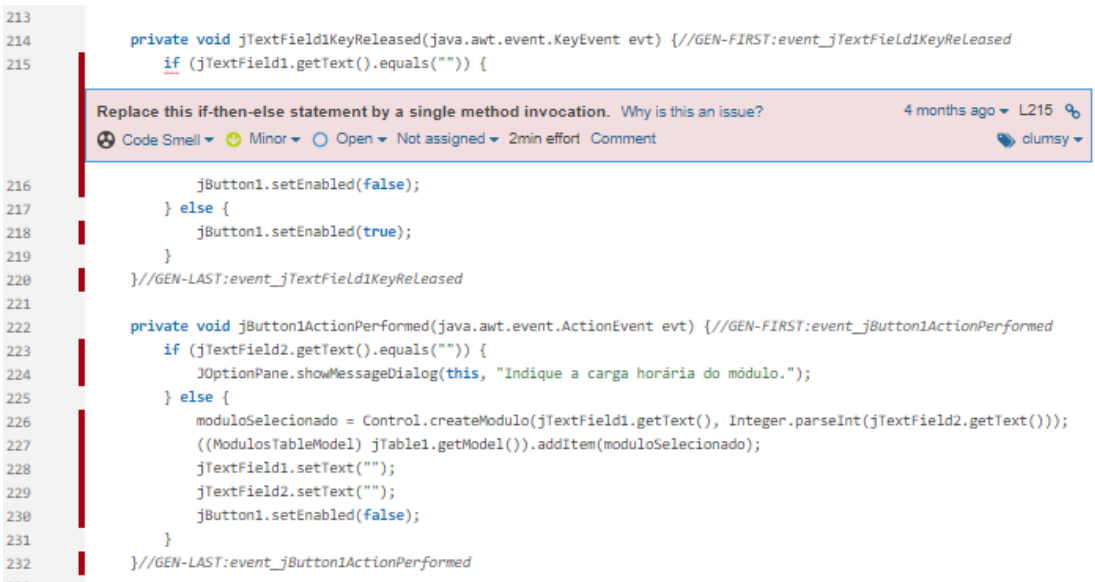


Figura 6.51 - Return of boolean expressions should not be wrapped into an "if-then-else" statement

Redundant casts should not be used

Camadas afetadas: Model e View

Expressões desnecessárias de conversão afetam negativamente a legibilidade do código.

Figura 6.52 - Redundant casts should not be used

Código Compatível:

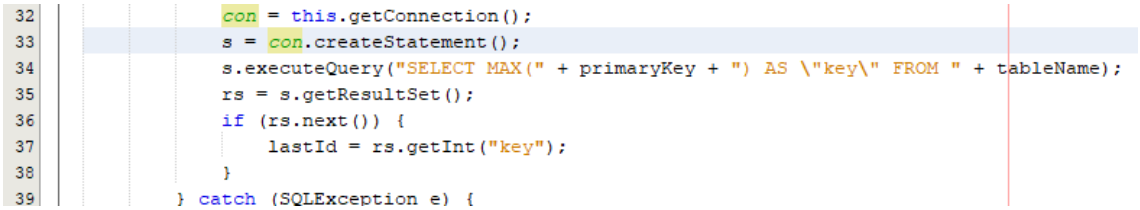


Figura 6.53 - Redundant casts should not be used Código Compatível

Boxed "Boolean" should be avoided in boolean expressions

Camada afetada: View

Quando um tipo em “caixa” java.lang.Boolean é utilizada como expressão, ele irá lançar o NullPointerException se o valor for nulo. É mais seguro evitar essa conversão e tratar o valor nulo explicitamente.

Figura 6.54 - Boxed "Boolean" should be avoided in boolean expressions

Multiple variables should not be declared on the same line

Camada afetada: Model

Declarar múltiplas variáveis na mesma linha de código pode dificultar a leitura.

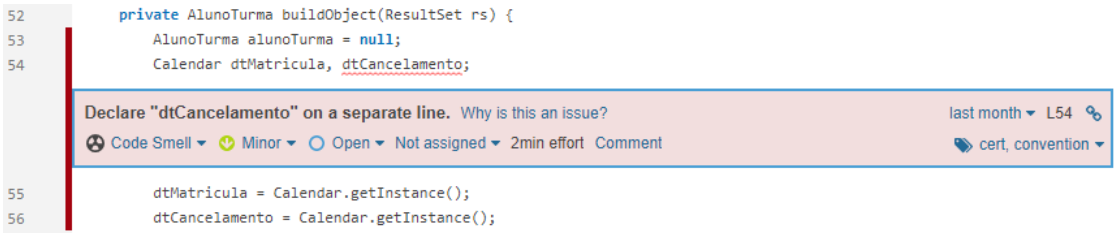


Figura 6.55 - Multiple variables should not be declared on the same line

Código compatível:

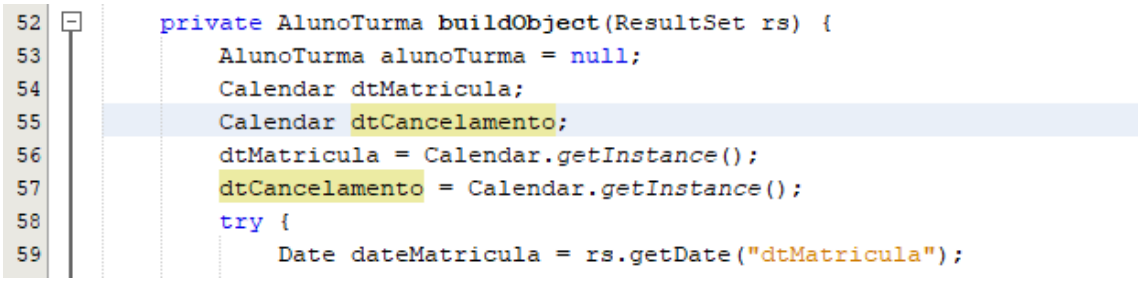


Figura 6.56 - Multiple variables should not be declared on the same line Código Compatível

Public constants and fields initialized at declaration should be "static final" rather than merely "final"

Camada afetada: View

Manter uma constante pública como apenas final, em vez de static final, implica na duplicação de seu valor a cada instância da classe, o que aumenta desnecessariamente a memória necessária para executar o programa.

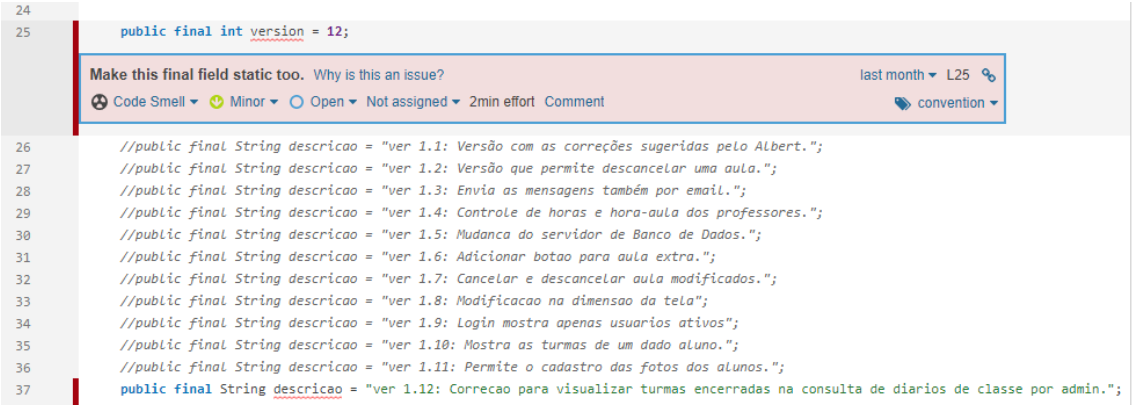


Figura 6.57 - Public constants and fields initialized at declaration should be "static final" rather than merely "final"

Código compatível:

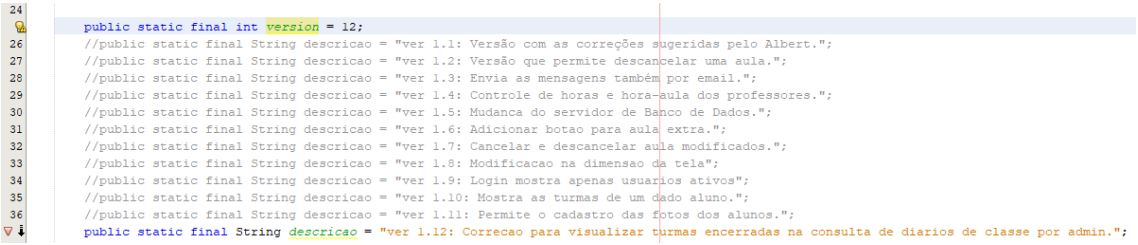


Figura 6.58 - Public constants and fields initialized at declaration should be "static final" rather than merely "final" Código Compatível

Unused local variables should be removed

Camadas afetadas: Model e View

Variáveis locais não utilizadas devem ser removidas. Isso ajuda na manutenibilidade do código, visto que evita que outros desenvolvedores tentem encontrar o uso da variável. Esse tópico está diretamente ligado ao code smell Dead Code.

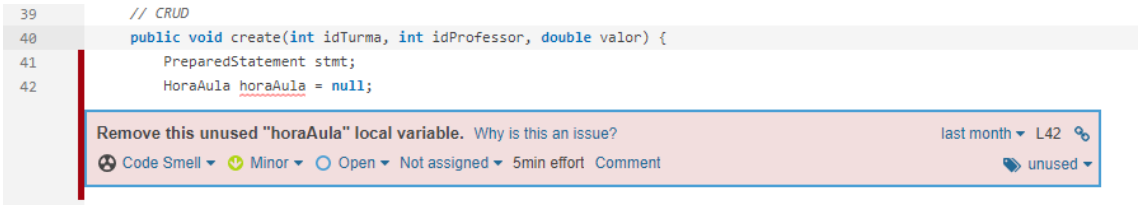


Figura 6.59 - Unused local variables should be removed

Código compatível:

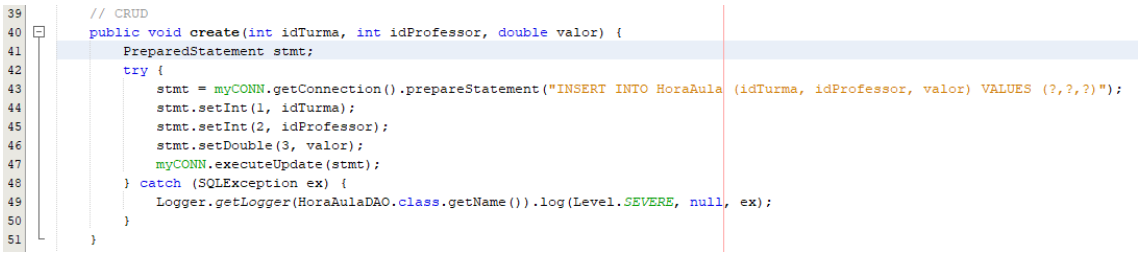


Figura 6.60 - Unused local variables should be removed Código Compatível

Switch statements should have at least 3 "case" clauses

O Switch é útil em casos com muitos casos dependentes do valor de uma mesma expressão.

Quando há menos de 2 casos, é recomendado manter o uso da instrução if, para que o código seja melhor legível.

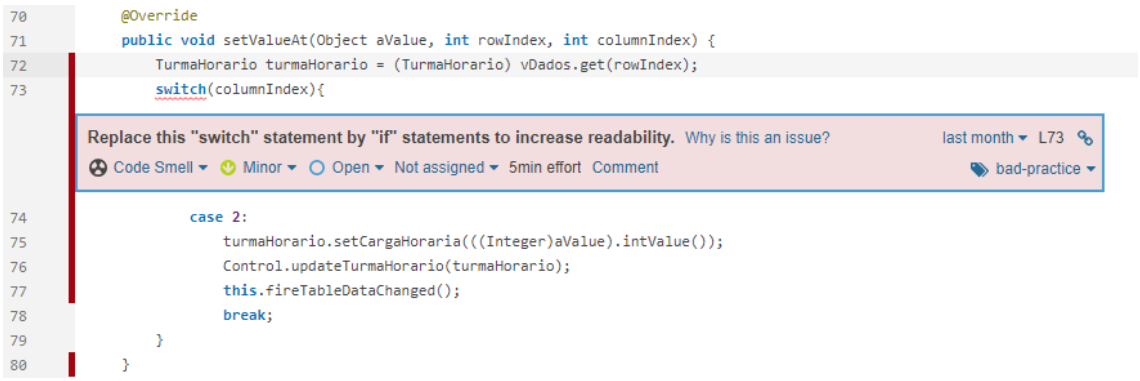


Figura 6.61 - Switch statements should have at least 3 "case" clauses

Código compatível:

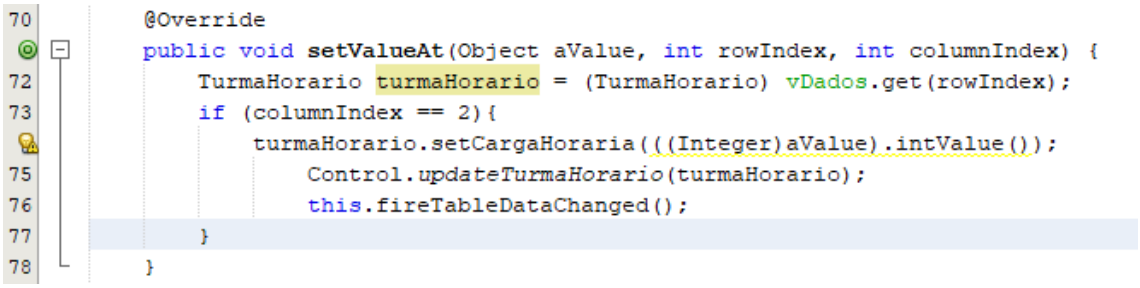


Figura 6.62 - Switch statements should have at least 3 "case" clauses Código Compatível

Throws declarations should not be superfluous

Camada afetada: Model

Uma exceção em uma declaração no Java se torna supérflua quando:

- É listada múltiplas vezes
- É uma subclasse de outra exceção listada
- É um “RuntimeException”, ou um de seus “filhos”
- É completamente desnecessário por conta da exceção declarada é incapaz de ser lançada

A exceção citada “java.io.FileNotFoundException” é uma subclasse de “java.io.IOException”, e portanto, não é necessária.

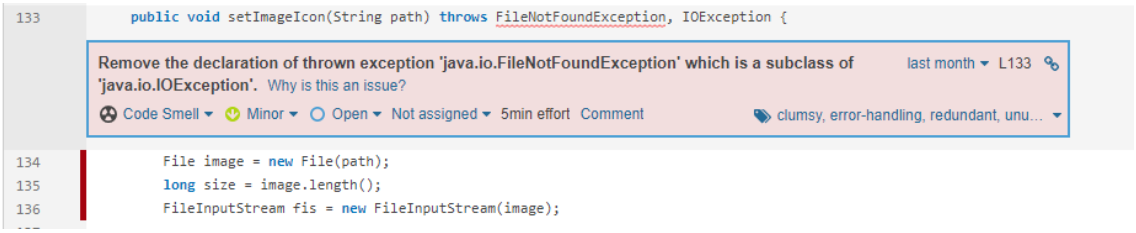


Figura 6.63 - Throws declarations should not be superfluous

Código compatível

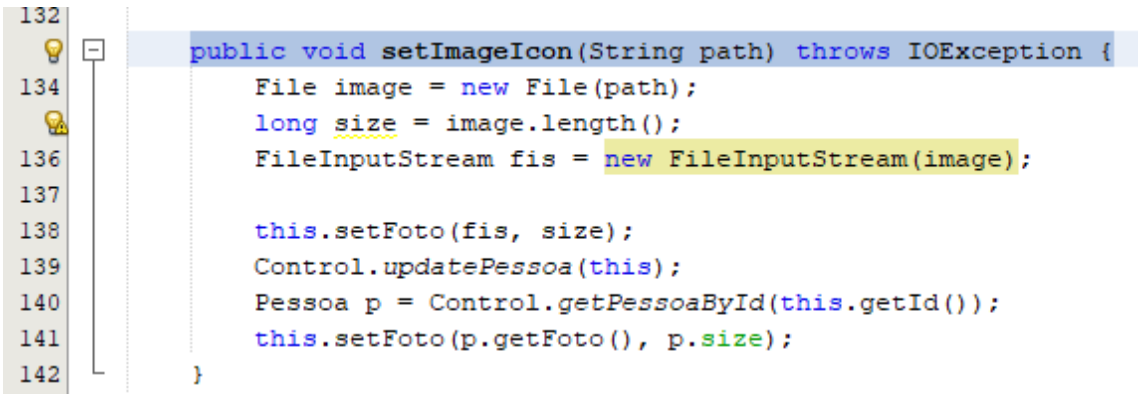


Figura 6.64 - Throws declarations should not be superfluous Código Compatível

Class variable fields should not have public accessibility

Camada afetada: Model

Campos de variáveis pertences à uma classe pública não respeitam o princípio de encapsulamento e possuem 3 desvantagens:

- Comportamentos adicionais como validação não podem ser adicionados
- A representação interna é exposta, e não pode ser modificada posteriormente
- Valores dos membros são sujeitos a mudanças em qualquer ponto do código e podem não satisfazer as suposições dos programadores.

Por meio do uso de atributos privados e métodos assessores (set e get), modificações não autorizadas são evitadas.

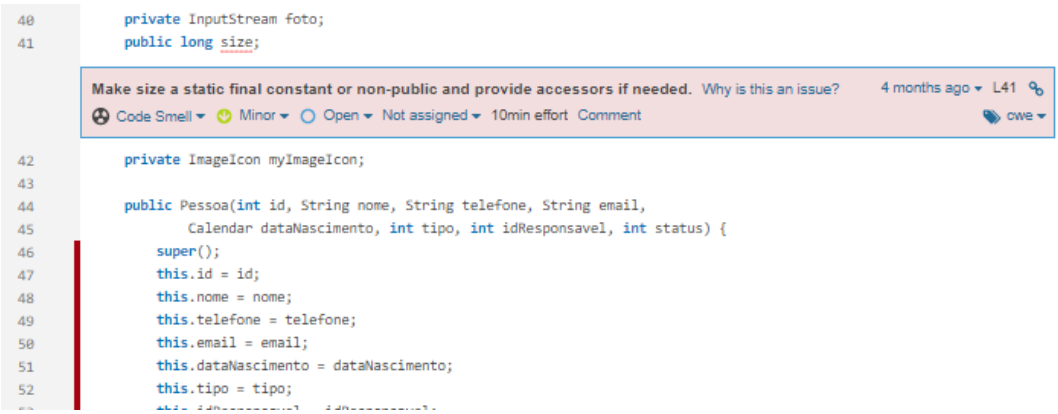


Figura 6.65 - Class variable fields should not have public accessibility

Local variables should not be declared and then immediately returned or thrown

Camada afetada: Control

Declarar uma variável e imediatamente retornar ou lançar o mesmo é uma má prática de programação.

Alguns desenvolvedores argumentam que essa prática melhora a legibilidade do código, visto que permite que nomeiem explicitamente o que está sendo retornado. Porém, essa variável é um detalhe interno de implementação que não é exposto aos chamadores do método. O nome do método deve ser o suficiente para que os chamadores saibam exatamente o que vai ser retornado.

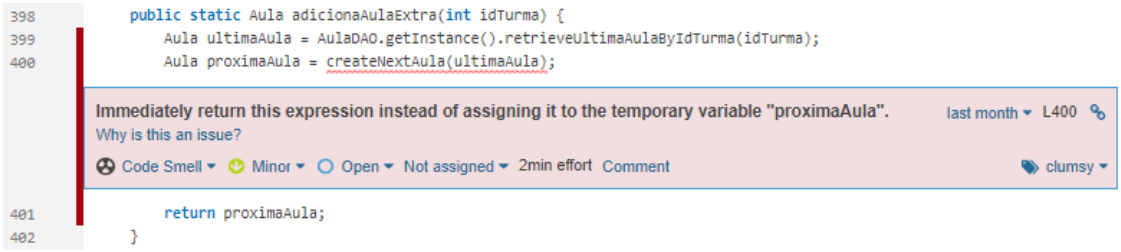


Figura 6.66 - Local variables should not be declared and then immediately returned or thrown

Código compatível:

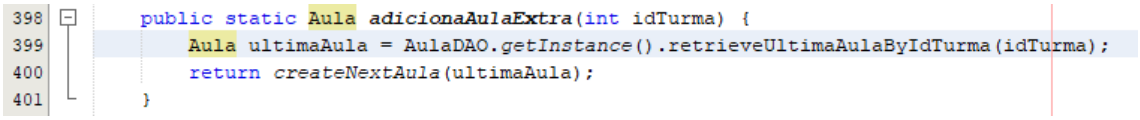


Figura 6.67 - Local variables should not be declared and then immediately returned or thrown Código Compatível

Static non-final field names should comply with a naming convention

Camada afetada: Model

As convenções de nomenclatura compartilhadas permitem que as equipes colaborem com eficiência. Esta regra verifica se os nomes de campo não finais estáticos correspondem a uma expressão regular fornecida.

Figura 6.68 - Static non-final field names should comply with a naming convention

Info

A análise inicial do SonarQube encontrou 1 tipo de regra violada, de severidade “info”, contio na tabela a seguir:

Rule	Quantity	Severity
Track uses of "TODO" tags	2	Info

Tabela 4.4 -

Track uses of "TODO" tags

Camada afetada: View

Tags “TODO” são comumente utilizadas para marcar locais onde mais código é necessário, mas que o desenvolvedor quer implementar posteriormente.

Algumas vezes, o desenvolvedor não possui tempo ou simplesmente esquece de voltar à tal tag.

Essa regra busca encontrar essas tags, para confirmar que não serão esquecidas.

Figura 6.69 - Track uses of "TODO" tags

Os campos marcados no código, indicam comentários realizados pelo desenvolvedor, e não possuem o intuito de uma tag TODO. A análise foi descartada.

Bugs

A análise do código apontou 3 regras infligidas, classificadas como bugs, dentre as severidades Blocker, Major e Minor.

Blocker

A análise inicial do SonarQube encontrou 1 tipo de regra violada, de severidade “blocker”, contida na tabela a seguir:

Rule	Severity	Quantity
Resources should be closed	Blocker	102

Tabela 6.4 – Blocker Bug X Quantidade

Resources should be closed

Camadas afetadas: Model e View

Conexões, fluxos, arquivos e outras classes que implementam a interface “closeable” ou sua superinterface “AutoCloseable” precisam ser fechadas após o uso. Além disso, essa chamada de encerramento deve ser feita em um bloco finally, caso contrário, uma exceção pode impedir a chamada de ser feita. Preferencialmente, quando a classe implementa “AutoCloseable”, o recurso deve ser criado utilizando o padrão “try-with-resources” e será encerrado automaticamente.

A falha em encerrar corretamente os recursos pode resultar em vazamentos de recursos, que podem acabar causando problemas no aplicativo.

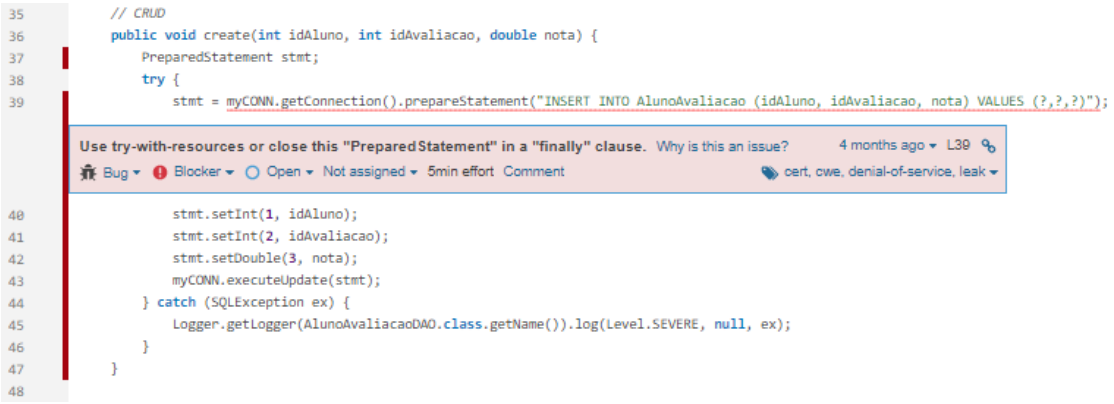


Figura 6.70 - Resources should be closed

Código Compatível:

```
38      public void create(int idAluno, int idAvaliacao, double nota) {
39          PreparedStatement stmt = null;
40          try {
41              stmt = myCONN.getConnection().prepareStatement("INSERT INTO AlunoAvaliacao (idAluno, idAvaliacao, nota) VALUES (?,?,?)");
42              stmt.setInt(1, idAluno);
43              stmt.setInt(2, idAvaliacao);
44              stmt.setDouble(3, nota);
45              myCONN.executeUpdate(stmt);
46          } catch (SQLException ex) {
47              Logger.getLogger(AlunoAvaliacaoDAO.class.getName()).log(Level.SEVERE, null, ex);
48          } finally{
49              try {
50                  if(stmt != null){
51                      stmt.close();
52                  }
53              } catch (SQLException ex2){
54
55              }
56          }
57      }
```

Figura 6.71 - Resources should be closed Código Compatível

Major

A análise inicial do SonarQube encontrou 1 tipo de regra violada, de severidade “major”, contida na tabela a seguir:

Rule	Severity	Quantity
Null pointers should not be dereferenced	Major	1

Tabela 6.5 – Major Bug X Quantidade

Null pointers should not be dereferenced

Camada afetada: View

Uma referência ao “null” não deve ser referenciada ou acessada. Isso faz com que o “NullPointerException” seja lançado. Na melhor das hipóteses, essa exceção irá causar o encerramento abrupto do programa. Na pior das hipóteses, ele pode expor informações de depuração que podem ser úteis à invasores ou permitir que um invasor contorne as medidas de segurança.

```
449     private void jTextField5MouseClicked(java.awt.event.MouseEvent evt) {//GEN-FIRST:event_jTextField5MouseClicked
450         JFrame f = new JFrame("Selecione o Responsável por " + (( 1 myPessoa != null) ? (myPessoa.getNome()) : ""));
451         f.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
452         ArrayList<Object> list;
453         if (jTextField5.getText().equals("")) {
454             list = Control.getAllPessoas();
455         } else {
456             list = new ArrayList<Object>();
457             list.add(Control.getPessoaById( 2 myPessoa.getIdResponsavel()));
        A "NullPointerException" could be thrown; "myPessoa" is nullable here.  Why is this an issue?   4 months ago ▾ L457
        Bug ▾  Major ▾  Open ▾  Not assigned ▾  10min effort  Comment                                   cert, cwe ▾
458         }
459         f.add(new PnlBuscaPessoas(list, myPessoa, f, jTextField5));
460         f.setSize(800, 500);
461         f.setVisible(true);
462     }//GEN-LAST:event_jTextField5MouseClicked
463
464     private void jButton3ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-FIRST:event_jButton3ActionPerformed
465         if (myPessoa != null) {
466             myPessoa.setIdResponsavel(0);
```

Figura 6.72 - Null pointers should not be dereferenced

Minor

A análise inicial do SonarQube encontrou 1 tipo de regra violada, de severidade “minor”, contida na tabela a seguir:

Rule	Severity	Quantity
Math operands should be cast before assignment	Minor	1

Tabela 6.6 – Minor Bug X Quantidade

Math operands should be cast before assignment

Camada afetada: View

Quando a aritmética é realizada em números inteiros, o resultado sempre será um número inteiro. Você pode atribuir esse resultado a um long, double ou float com conversão automática de tipo, mas tendo começado como um int ou long, o resultado provavelmente não será o esperado. Por exemplo, se o resultado da divisão interna for atribuído a uma variável de ponto flutuante, a precisão terá sido perdida antes da atribuição. Da mesma forma, se o resultado da multiplicação for atribuído a um longo, ele pode já ter transbordado antes da atribuição. Em qualquer dos casos, o resultado não será o esperado. Em vez disso, pelo menos um operando deve ser convertido ou promovido para o tipo final antes que a operação ocorra.

```
2693    private void jButton26ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-FIRST:event_jButton26ActionPerformed
2694        Pessoa professor = (Pessoa) jComboBox15.getSelectedItem();
2695        int mes = Integer.parseInt(jTextField9.getText());
2696        int ano = Integer.parseInt(jTextField10.getText());
2697        if (professor != null) {
2698            ArrayList<Object> vAulas = Control.getAllAulasDadasByIdProfessorMes(professor.getId(), mes, ano);
2699            int cargaHorariaTotal = 0;
2700            double valorTotal = 0;
2701            for (Object o : vAulas) {
2702                Aula aula = ((Aula) o);
2703                HoraAula horaAula = Control.getHoraAulaByIdTurmaIdProfessor(aula.getIdTurma(), aula.getIdProfessor());
2704                if (horaAula != null) {
2705                    valorTotal += (aula.getCargaHoraria() / 60) * horaAula.getValor();
```

Cast one of the operands of this integer division to a "double". Why is this an issue? 4 months ago L2705

Bug Minor Open Not assigned 5min effort Comment cert, cwe, overflow, sans-top25-risky

```
2706                }
2707                cargaHorariaTotal += aula.getCargaHoraria();
2708            }
2709            jTextField8.setText((cargaHorariaTotal / 60) + "");
2710            jTextField12.setText(valorTotal + "");
2711            ((RelatorioAulasDadasTableModel) jTable9.getModel()).addListOfItems(vAulas);
2712        }
```

Figura 6.73 – Math operands should be cast before assignment

Vulnerabilities
A análise do código apontou 1 regra infligida, classificada como vulnerabilidade, e com a severidade blocker:

Blocker
A seguinte regra violada foi encontrada pela análise do SonarQube:

Rule	Severity	Quantity
Databases should be password-protected	Blocker	1

Tabela 6.7 – Blocker Vulnerability X Quantidade

Databases should be password-protected

Camada afetada: Model

Banco de dados devem sempre ter as senhas protegidas. O uso de uma conexão ao banco de dados com uma senha vazia é um indício de que o banco de dados não está protegido.

```
29      @Override
30      public Connection getConnection() {
31          while (con == null) {
32              try {
33                  //Class.forName(DRIVER).newInstance();
34                  con = DriverManager.getConnection(DBURL, user, pass);
                Add password protection to this database. Why is this an issue?   4 months ago ▾ L34
                Vulnerability ▾ Blocker ▾ Open ▾ Not assigned ▾ 45min effort Comment   cwe, owasp-a3 ▾
35              } catch (Exception e) {
36                  //JOptionPane.showMessageDialog(null, "Não consegui acessar o banco em: "+DBURL+"\nVou tentar localmente.");
37                  String[] choices = {"Sim, tenho um banco de dados local.", "Não, apenas encerre."};
38                  int resp = JOptionPane.showOptionDialog(
39                          null // Center in window.
40                          , "Não consegui acessar o banco em:\n"+DBURL+"\nDevo tentar o banco local?" // Message
41                          , "Acesso ao Banco de Dados" // Title in titlebar
42                          , JOptionPane.YES_NO_OPTION // Option type
43                          , JOptionPane.PLAIN_MESSAGE // messageType
```

Figura 6.74 - Databases should be password-protected

Security Hotspots

A análise do código apontou 4 regras infligidas, classificadas como security hotspots, dentre as severidades High e Low.

High

A análise inicial do SonarQube encontrou 3 tipos de regras violadas, de severidade “High”, contidas na tabela a seguir:

Rule	Severity	Quantity	Type
Ensure that string concatenation is required and safe for this SQL query	High	19	SQL Injection
'password' detected in this expression, review this potentially hard-coded credential.	High	1	Authentication
'PWD' detected in this expression, review this potentially hard-coded credential.	High	1	Authentication

Tabela 6.8 – High Security Hotspot X Quantidade

Ensure that string concatenation is required and safe for this SQL query

Camada afetada: Model

As consultas SQL geralmente precisam usar uma string SQL codificada com um parâmetro dinâmico proveniente de uma solicitação do usuário. Formatar uma string para adicionar esses parâmetros à solicitação é uma prática ruim, pois pode resultar em uma injeção de SQL. A maneira segura de adicionar parâmetros a uma consulta SQL é usar mecanismos de vinculação SQL.

```
72      public ArrayList<Curso> retrieveLike(String nome) {
73          PreparedStatement stmt;
74          ArrayList<Curso> cursos = new ArrayList<Curso>();
75          ResultSet rs;
76          try {
77              stmt = myCONN.getConnection().prepareStatement("SELECT * FROM Curso WHERE nome LIKE '%"+nome+"%' ORDER BY nome");
78              rs = myCONN.getResultSet(stmt);
79              while (rs.next()) {
80                  cursos.add(buildObject(rs));
81              }
82              rs.close();
```

Figura 6.75 – Ensure that string concatenation is required and safe for this SQL query

'PWD' detected in this expression, review this potentially hard-coded credential.

Camada afetada: Model

Como é fácil extrair strings de um código-fonte ou binário do aplicativo, as credenciais não devem ser codificadas permanentemente. Isso é particularmente recomendado para aplicativos distribuídos ou de código-fonte aberto.

Figura 6.76 - 'PWD' detected in this expression, review this potentially hard-coded credential.

Password detected in this expression, review this potentially hard-coded credential.Add CommentOpen in IDE

Camada afetada: Model

Como é fácil extrair strings de um código-fonte ou binário do aplicativo, as credenciais não devem ser codificadas permanentemente. Isso é particularmente recomendado para aplicativos distribuídos ou de código-fonte aberto.

Figura 6.77 - Password detected in this expression, review this potentially hard-coded credential.Add CommentOpen in IDE

Low

A análise inicial do SonarQube encontrou 1 tipo de regra violada, de severidade “Low”, contidas na tabela a seguir:

Rule	Severity	Quantity	Type
Make sure this debug feature is deactivated before delivering the code in production.	Low	16	Insecure Configuration

Tabela 6.9 -Low Security Hotspot X Quantidade

Make sure this debug feature is deactivated before delivering the code in production

Camada afetada: Model

A entrega de código em produção com recursos de depuração ativados é uma questão de segurança.

Figura 6.78 - Make sure this debug feature is deactivated before delivering the code in production