Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Sistemas de Informação

Aplicação de técnicas de engenharia de *software* para identificação e melhorias no processo de manutenção de código-fonte

Julio Cesar Rosa Trindade

JUIZ DE FORA
JUNHO, 2023

Aplicação de técnicas de engenharia de *software* para identificação e melhorias no processo de manutenção de código-fonte

Julio Cesar Rosa Trindade

Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Sistemas de Informação

Orientador: Marco Antônio Pereira Araújo

JUIZ DE FORA
JUNHO, 2023

Resumo

Para automatizar e garantir melhorias de processos, corporações investem recursos para criação de projetos de softwares. Adicionalmente, dependendo de seu contexto, estes softwares podem ter grande orçamento e equipes de desenvolvimento quando são essenciais para a corporação, porém, caso não sejam, o orçamento e mão-de-obra empregados na construção destes acaba por não viabilizar tempo e recursos necessários para seu bom desenvolvimento e manutenção. Com isso, existe a possibilidade de projetos internos se tornarem demasiadamente complexos por não observarem os princípios da engenharia de softwares desde que são concebidos. Este trabalho aplica ferramentas de análise de qualidade de código-fonte em um projeto do Departamento de Ciência da Computação da Universidade Federal de Juiz de Fora e, dadas as métricas encontradas, cria abordagens para solução dos problemas apontados. Adicionalmente, cria estratégias que podem ser aplicadas para que o projeto possa ser mantido e evoluído o mais próximo possível das melhores práticas de desenvolvimento. Em adição, demonstra as consequências da não adoção de técnicas de engenharia de software na tentativa de aplicação tardia da mesma.

Keywords: engenharia de software; sonarqube; orientação a objetos; teste de unidade; documentação de software;

Agradecimentos

Aos meus amigos, que estiveram presentes durante todos os percalços na produção deste trabalho, especialmente à Raiza, que lia e relia as versões que eu a encaminhava, ajudando-me a corrigir possíveis erros. Agradeço ao Jobson e à Julia, que me motivaram o tempo todo e não me deixaram desistir (do mesmo modo que vou motivá-los a se casar). À Taís e à Ellen, que me incentivaram e acreditavam que seria possível.

Aos meus professores e coordenadores, especialmente ao Luciano Jerez Chaves e ao meu orientador, Marco Antônio Pereira Araujo, que tiveram a paciência que certamente me faltaria no lugar deles.

Agradeço à minha família, que demonstrou seu apoio de uma maneira bem peculiar.

E, por fim, à aleatoriedade do destino, que alguns chamam de Deus, mas que não gosto de definir assim. Mesmo assim, agradeço por ter permitido que, de uma forma curiosa, tudo acontecesse de modo a tornar possível este momento.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

OO	Orientação a objetos
CI	*Continuous integration*
CD	*Continuous delivery / Continuous deployment*
PR	Pull request
TI	Tecnologia da informação
DCC	Departamento de Ciência da Computação
MAT	Departamento de Matemática
ICE	Instituto de Ciências Exatas
UFJF	Universidade Federal de Juiz de Fora
IDE	*Integrated development environment*
HTML	*Hyper Text Markup Language*
CSS	*Cascade Style Sheet*
JS	*Javascript*
P0	Passo 0 do trabalho
P1	Passo 1 do trabalho
DI	*Dependency injection*
IoC	*Inversion of control*
SOAP	*Simple Object Access Protocol*
AFF	*Architectural Fitness Function*
SIGA	Sistema Integrado de Gestão Acadêmica
ACE	Atividade curricular eletiva
IRA	Índice de rendimento acadêmico

1 Introdução

Ao longo dos anos, departamentos de tecnologia da informação (TI) foram criados nas mais diversas corporações para atender diversas demandas de modernização. Estes, utilizam a informática como ativdade meio para atingir seus fins da forma mais eficiente possível. Estes, por diversas vezes, são responsáveis pela criação e implantação de Sistemas de Informação (SI) dentro destas corporações. Como define Laurindo et al. (2001), TI pode ser considerado o aspecto técnico e SI pode ser relativo ao fluxo de trabalho, pessoas e informações envolvidas. Os SI's visam oferecer melhorias de gerenciamento, ganho de produtividade e competitividade nos negócios (FRITOLA; SANTANDER, 2021).

Para a criação adequada de um SI, tanto processos de engenharia de requisitos (ER) quanto processos de engenharia de software (ES) devem ser observados. A ER visam representar os objetivos dos interessados no projeto, também chamados de *stakeholders* (FRITOLA; SANTANDER, 2021), enquanto a ES visa o processo de desenvolvimento de um software, buscando padronização e qualidade do produto final (SANTOS GUILHERME VILATORO E LOPES, 2020).

Em linhas gerais, é possível englobar ER e ES dentro da área de Governança de TI. A Governança de TI, como descreve (TAROUCO; GRAEML, 2011), permite que as perspectivas de negócios, de infraestrutura de pessoas e de operações sejam levadas em consideração no momento de definição das ações de TI.

Para este trabalho, empiricamente, os projetos de TI serão classificados como sendo (i) projetos de impacto aos clientes e consumidores de uma corporação, (ii) projetos de impacto administrativo, gerencial e operacional da corporação.

Os projetos de impacto aos clientes e consumidores de uma corporação são aqueles onde o sistema representa o principal serviço prestado pela corporação. Estes projetos podem causar impactos gigantescos à credibilidade das corporações e gerar diversas consequências legais, como processos judiciais e administrativos. Normalmente, este tipo de projeto possui gestão de TI adequada, o que resulta em maior orçamento, planejamento e análise.

Por outro lado, os projetos caracterizados como de impacto administrativo, gerencial e operacional, surgem para gerar melhorias na execução de demandas internas, muitas das vezes não possuem orçamento e planejamento adequados destinados a eles, o que pode prejudicar todo seu ciclo de vida, especialmente no aspecto de qualidade e evolução. Estes projetos podem ser caracterizados como possuindo baixo nível gestão de TI.

1.1 Descrição do problema

A Universidade Federal de Juiz de Fora, como corporação, possui diversos sistemas destinados aos seus alunos e professores. Alguns destes sistemas são amplamente usados a nível de corporação, como o Sistema Integrado de Gestão Acadêmica (SIGA) e o Moodle (plataforma utilizada para gestão de cursos à distancia). No SIGA, alunos, professores e coordenadores de curso podem exercer operações e processos importantes relativos às suas competências. Estas operações e processos, por sua vez, são realizados de forma unificada pela instituição. Já para os departamentos, como o Departamento de Ciência da Computação (DCC) ou o Departamento de Matemática (MAT), estas operações e processos podem conter diferenças significativas, o que motiva seus integrantes a idealizarem e desenvolverem sistemas para otimizar suas operações internas. Como não existe centralização das equipes de TI que executam estes projetos, não é garantido que boas práticas de programação são aplicadas.

Para este trabalho, usaremos ferramentas previstas na Engenharia de Software para obter métricas de qualidade sobre um destes sistemas departamentais, chamado "Sistema de Gestão de Curso", criado pelo DCC para prover informações relevantes aos coordenadores, docentes e discentes em forma de tabelas, gráficos e uma ferramenta para auxiliar o planejamento de formatura. Este caso se enquadra no tipo de projetos de impacto administrativo, gerencial e operacional da corporação.

1.2 Motivação

Em entrevista com Luciano Jerez Chaves, coordenador do curso de Sistemas de Informação à época e um dos principais usuários do sistema, o sistema "Sistema de gestão de curso" apresenta *bugs*, que não são mapeados em nenhum sistemas/plataformas de gestão de incidentes e de progressos feitos no sistema. Segundo o mesmo, o sistema encontra-se "abandonado" por volta de três anos e apenas alguns alunos fizeram correções de erros, mas nenhum tipo de melhoria. Também não existe documentação adequada e nem mesmo informações sobre ambientes para execução do sistema. Estes dados podem ser vistos no .1.

Baseado nos relatos acima mencionados, pode-se inferir que este sistema não possui gestão de TI adequada, o que abre espaço para pesquisas exploratórias sobre a qualidade do sistema como um todo. É possível fazer uso de ferramentas de análise de código-fonte para identificar possíveis problemas e, eventualmente, corrigí-los, melhorando assim a qualidade técnica do sistema.

1.3 Objetivos

O objetivo deste trabalho é aplicar técnicas de Engenharia de Software para identificar e corrigir eventuais falhas provenientes de processo de desenvolvimento sem gestão de TI adequada. Deste objetivo, podemos listar os objetivos específicos abaixo:

- Executar análise no código-fonte do sistema alvo, neste caso o sistema *Sistema de Gestão de Curso*.
- Extrair métricas de *bugs*, analisando quais deles são mais críticos para a aplicação no estado atual do desenvolvimento.
- Extrair métricas de cobertura de teste de unidade no estado atual do desenvolvimento.
- Extrair métricas de duplicação de código-fonte no estado atual do desenvolvimento.

- Criar mecanismos de análise contínua para acompanhamento da evolução do trabalho e de futuras evoluções do código-fonte.

- Definir metodologias para resolução de eventuais problemas encontrados.

- Construir documentação técnica para configurações, definições e análises de aspectos relevantes na evolução do sistema.

2 Referencial teórico

A TI evoluiu de uma orientação tradicional de suporte administrativo para um papel estratégico dentro da organização. A visão da TI como arma estratégica competitiva tem sido discutida e enfatizada, pois não só sustenta as operações de negócio existentes, mas também permite que se viabilizem novas estratégias empresariais (LAURINDO et al., 2001).

A produção de softwares por departamentos de TI possibilitou que mudanças dos mais variáveis portes pudessem acontecer dentro das corporações. Nos dias atuais, processos que outrora podiam ser realizados apenas de forma manual, como efetuar operações bancárias, tornaram-se possíveis pela evolução tecnológica e pelo investimento das empresas em produção de sistemas (LAURINDO et al., 2001). As vantagens da implantação destes artefatos de tecnologia beneficia ambos os lados da interação cliente-corporação.

Como menciona Beal (2001), o principal benefício que a tecnologia traz para as organizações é a sua capacidade de melhorar a qualidade e a disponibilidade de informações e conhecimentos para a empresa, seus clientes e fornecedores.

Contudo, tanto Beal (2001) quanto Laurindo et al. (2001) pontuam que estes projetos devem ser cuidadosamente geridos. Uma má gestão de software pode ocasionar em sérios prejuízos, que podem ser relacionados à imagem e credibilidade de uma corporação ou relacionados à custos de evolução e manutenção dos sistemas desenvolvidos. Para este trabalho, iremos nos aprofundar nos aspectos técnicos da gestão.

Para uma gestão técnica eficaz, o ideal é que planejamentos e projetos relacionados a softwares sejam feitos antes mesmo da etapa de codificação (SANTOS GUILHERME VILATORO E LOPES, 2020). Dessa forma, é possível estabelecer métodos para uma boa evolução técnica e ferramentas de métricas para garantir que estes métodos estão sendo aplicados corretamente. Do mesmo modo, documentações devem ser geradas para referências futuras. A documentação de um projeto é capaz de causar efeitos positivos para entendimento, configurações e manutenções futuras, como observado no trabalho de (FRITOLA; SANTANDER, 2021).

A área responsável por convencionar principios de engenharia englobando aspéctos técnicos e não-técnicos para produção de softwares de qualidade de forma eficaz é a Engenharia de software (VASCONCELOS; ROUILLER; AND, 2006). A engenharia de software propõe uma série de técnicas para que um projeto possa ser criado, evoluído e mantido com uma taxa de integridade estrutural e desempenho satisfatórios. Essas técnicas são chamadas de boas práticas de programação.

O uso de boas práticas tende a tornar o código de um desenvolvedor mais fácil de se entender e manuser por outros desenvolvedores, além de oferecer uma boa taxa de evolutividade e manutenabiliade deste (JESUS, 2016).

Segundo Arakaki e Ferreira (2017), o esforço de manutenção de um software deve ser adequado às necessidades de mudanças que podem ocorrer devido à sua aplicação real, que são de dificil previsibilidade, já que podem acontecer a qualquer momento e por qualquer motivo. Porém, o esforço e complexidade para se analisar os mais diversos aspectos envolvidos na qualidade de código de um sistema é alto.

Considerando que sistemas internos tendem a possuir equipes reduzidas de desenvolvedores, torna-se necessário o uso de ferramentas que possam fazer este tipo de verificação da forma mais automatizada possível. (FOWLER, 2006) descreve CI como possibilidades de centralização de código-fonte, automatização de compilação, execução de testes e análises e facilitação de observação dos critérios de qualidade. De acordo com Assunção (2021), a adoção de ferramentas que possam auxiliar nesta tarefa é importante para a detecção de indícios de problemas na engenharia e arquitetura do sistema.

Em adição, Meirelles (2013) complementa que existem particularidades nas métricas de software. Uma métrica de cobertura de testes de unidade não é capaz de revelar problemas relacionados à manutenabilidade, modularidade, flexibilidade ou mesmo simplicidade de implementações. Por este motivo, devemos ter clareza no entendimento das métricas que serão utilizadas para considerar o código de um sistema bom ou ruim. Essas métricas, por sua vez, demandam certo conhecimento por parte dos analisadores, que muitas das vezes também são desenvolvedores. Estes, conhecendo aspectos de engenharia de software, podem caracterizar as métricas geradas como mais ou menos relevantes.

Analisando a literatura até este ponto, já temos arcabouço teórico para entender

os benefícios da construção de um processo de CI que compila, testa e analisa a aplicação a cada incremento de código-fonte desenvolvido para evitar evitar agravamentos e indicar possíveis direções para melhorias. No estudo realizado por Neto (2021), é possível identificar os beneficios que projetos que implementam CI podem oferecer quando comparados com projetos que não implementam tal técnica, especialmente no quesito de cobertura de teste de unidade. Fowler (2006) menciona que, embora CI não tenha a capacidade de evitar nem corrigir *bugs*, aumenta dramaticamente a visibilidade, facilitando assim que tanto gestão quanto desenvolvedores tomem atitudes.

O Sonarqube, como ferramenta de análise, oferece essas estimativas de forma clara, provendo informações relevantes para cada uma das vulnerabilidades apontadas (TOMOMITSU, 2021).

No sistema "Sistema de gestão de curso", diversos profissionais com pouca ou nenhuma experiência fizeram alterações em estruturas importantes do código-fonte, não observando questões importantes como testabilidade, acoplamento e complexidade. Magalhães, Junior e Araújo (2018) discorre sobre razões pelas quais desenvolvedores, sobretudo iniciantes, podem acabar se preocupando, na sua maioria, com códigos funcionais em detrimento de códigos de teste de unidade.

Empiricamente, podemos mencionar os impactos da senioridade dos integrantes do projeto na qualidade do *código-fonte*. (SILVA, 2020) e (ALFAYEZ et al., 2018), em seus estudos, mencionam que a senioridade pode ser composta por diversos fatores, como quantidade e periodicidade de *commits* e tempo de atuação no projeto. Hipoteticamente, quanto mais tempo um desenvolvedor atua num projeto, mais experiente ele é. Adicionalmente, é possível adicionar o conhecimento em engenharia de software como quesito de senioridade neste ponto, pois as análises podem relevar fragilidades relacionadas à principios e boas práticas não aplicadas, mesmo com alto índice de entregas. Se os desenvolvedores não possuem base para implementação dos padrões de projetos necessários, é possível que o sistema apresente fragilidades estruturais (BENTO, 2020). Estas fragilidades estruturais podem comprometer a aplicação de testes de unidades, por tornar esta atividade consideravelmente mais complexa, gerando novos *bugs* e comprometendo assim a credibilidade e aderência de um sistema por seus usuários (OLIVEIRA; DARCE, 2013).

O alto acoplamento e acúmulo de responsabilidades pode oferecer desafios para implementação de testes de unidade. Bento (2020) pontua que "*designs* de código que têm classes que carregam mais de uma responsabilidade cheiram à fragilidade". Esta afirmação tem fundamento quando utilizamos o SOLID como base. O SOLID é o acrônimo usado para denotar cinco princípios que auxiliam o desenvolvimento de sistemas. Gonçalves (2022) menciona que estes princípios tornam o código resiliente, mais simples e viabiliza componentização e interoperabilidade por outros sistemas. Neste trabalho serão abordados os princípios de responsabilidade única, representado pela letra "S" e o princípio da injeção de dependência (DI), representado pela letra "I". A injeção de dependência oferece vários benefícios para o desenvolvimento de sistemas, especialmente em testes de unidade. Utilizando injeção de dependência é possível criar testes de forma mais flexível. Fowler (2005) define a injeção de dependência como sendo a possibilidade de prover uma funcionalidade a uma classe sem a necessidade desta ter detalhes de como a funcionalidade foi implementada, mas sim apenas confiando na interface provida. Utilizando DI aliado com técnicas de objetos *Fake* para simular comportamentos. Os objetos *Fake*, como descreve Gonçalves (2022), são representações ou unidades falsas que simulam as ações de unidades reais.

Todas as possibilidades descritas anteriormente acabam por diminuir a qualidade do projeto como um todo, principalmente quando esses sistemas são considerados legados. Fritola e Santander (2021) descreve como sistemas legados os sistemas antigos que permanecem em operação nas organizações e que comumente oferecem uma alta complexidade, dificil manutenção e configuração e pouca ou nenhuma documentação, seja técnica ou de negócio.

3 O sistema 'Sistema de Gestão de Curso'

O sistema "Sistema de gestão de curso" foi desenvolvido para oferecer informações úteis para as coordenações dos cursos do Instituto de Ciências Exatas (ICE) possam, de forma mais eficiente, gerenciar aspectos importantes dos cursos, conforme podemos observar na Figura 3.1.

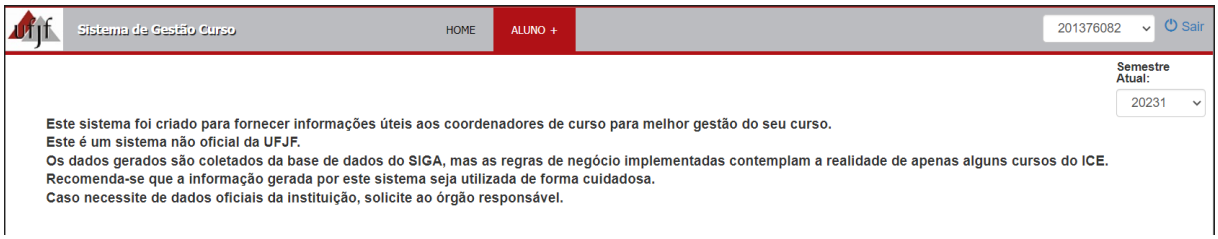

Figura 3.1: Sistema de gestão de curso - Tela inicial

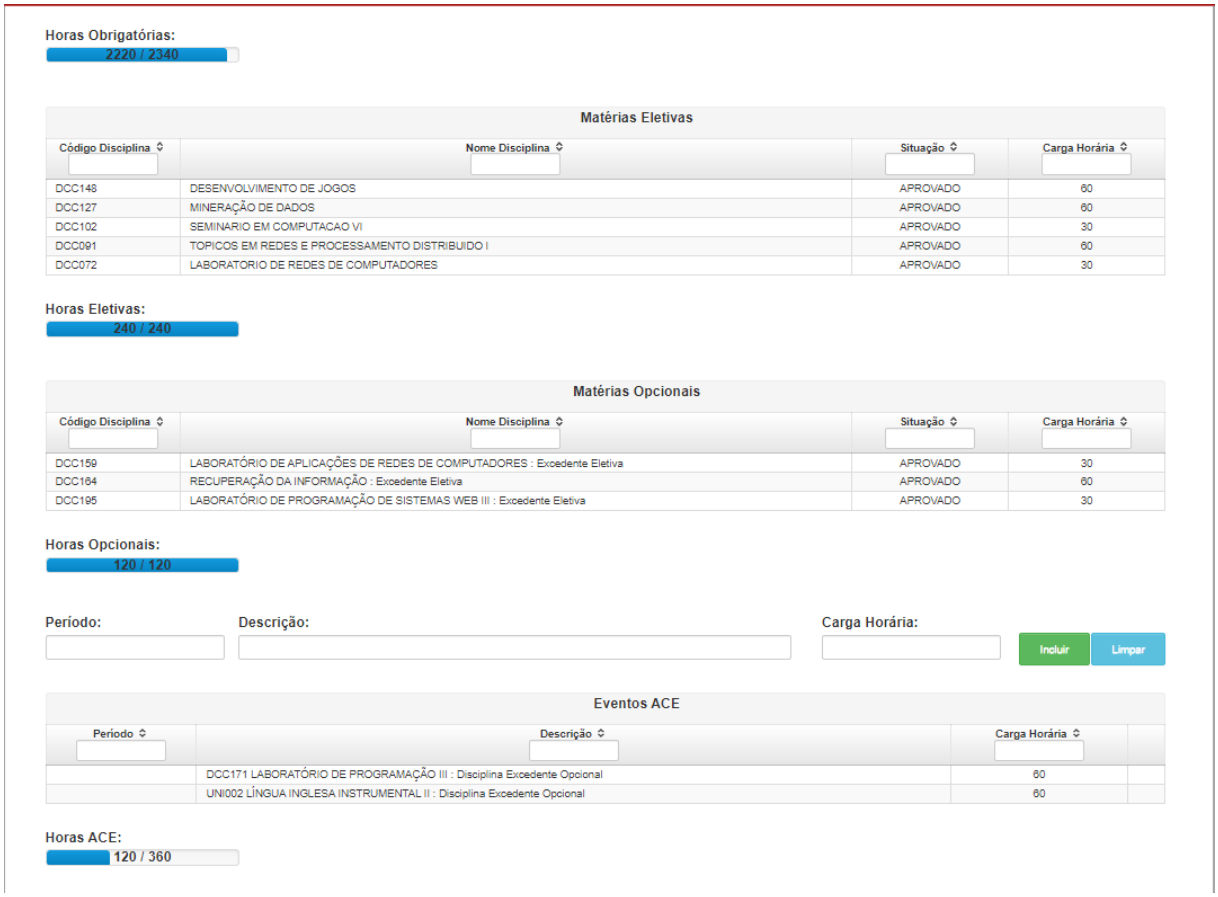

Figura 3.2: Sistema de gestão de curso - Situação do Aluno

O sistema utiliza dados curriculares dos alunos, que são obtidos através de inte-

gração com o SIGA, para gerar relatórios sobre situação do aluno, como pode ser visto na Figura 3.2, percentual de curso concluído, quantidade de horas obrigatórias, eletivas, opcionais e de atividade curricular eletiva (ACE) concluídas e faltantes. Também oferece gráficos de visualização de índice de rendimento acadêmico (IRA) por período e acumulado, conforme Figura 3.3.

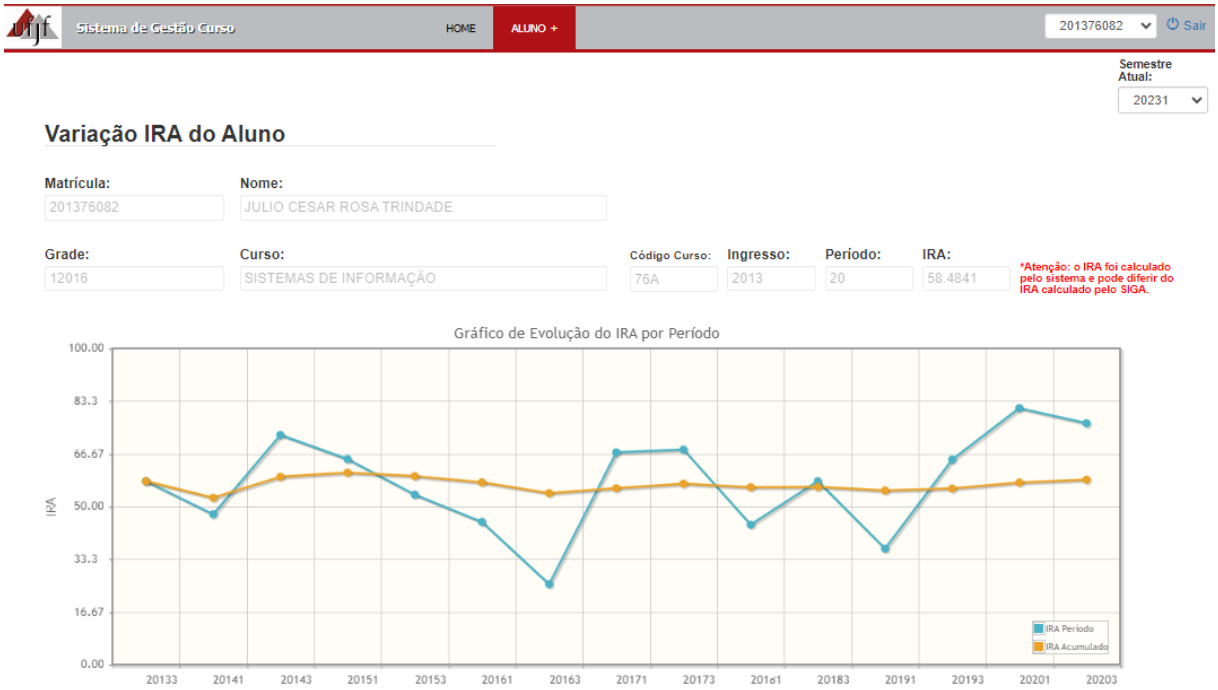

Figura 3.3: Sistema de gestão de curso - IRA

Além destas funcionalidades, o sistema oferece a função de planejamento de formatura. Utilizando os dados curriculares da aprovação, restrições de pré-requisitos, co-requisitos e oferta de disciplinas por período é possível montar, período a período, uma grade de curso sugerida para que o aluno consiga planejar o que irá estudar até cumprir todos os requisitos para sua formatura, como ilustrado na Figura 3.4.

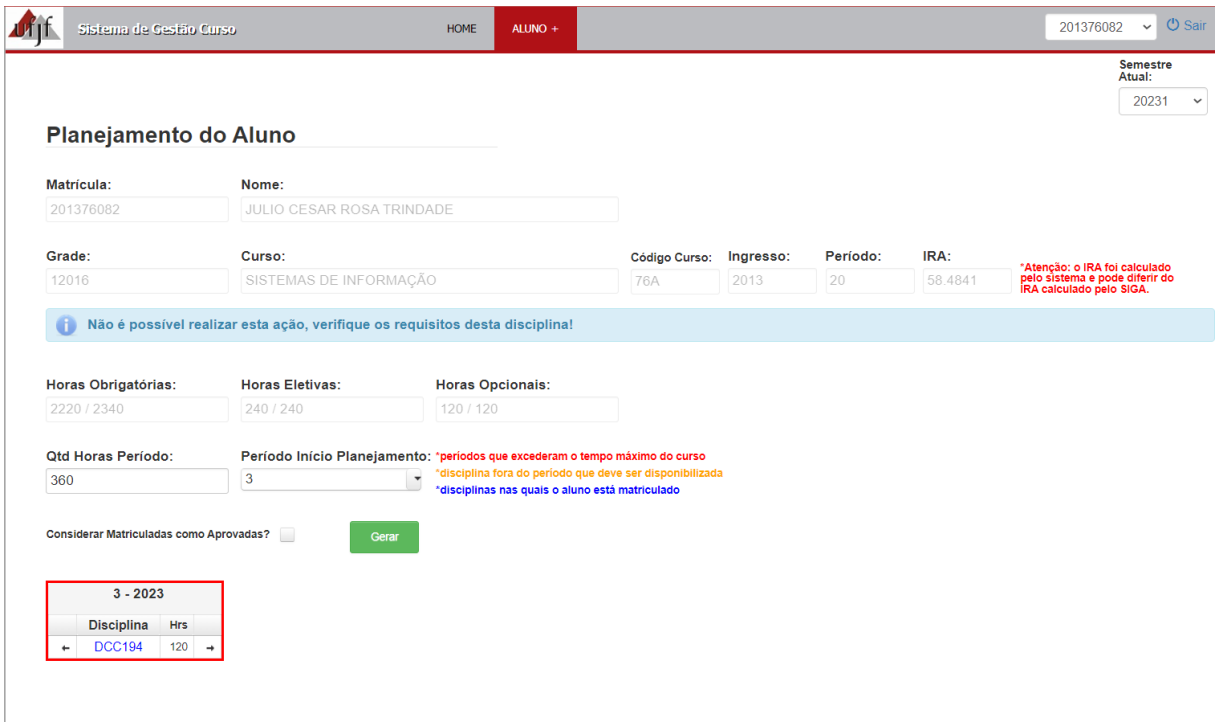

Figura 3.4: Sistema de gestão de curso - Planejamento de formatura

4 Ferramentas e Tecnologias

Este capítulo visa oferecer detalhes sobre as tecnologias, ferramentas e conceitos utilizados para atingir os objetivos deste trabalho. As subseções dentro de cada seção mencionam ítens de relevância dentro de cada escopo.

4.1 Sonarqube

O Sonarqube é uma ferramenta construída para efetuar análises de código-fonte e amplamente utilizada em ambientes corporativos (S.A, 2023). Baseado em cada linguagem de programação, esta ferramenta tem a capacidade de apontar diversos *bugs* e indícios de problemas de desenvolvimento, comumente chamados de *code smells*, classificando-os por severidade. Além disso, tem a capacidade de gerar métricas de cobertura de testes de unidade, duplicação de código-fonte e é amplamente configurável, viabilizando a criação de perfis adicionais de análise, definidos pelos usuários. O Sonarqube é usado em diversos cenários de desenvolvimento de sistemas. Utilizando uma das mais de trinta linguagens de programação suportadas por esta ferramenta e a ampla documentação disponível *online*. Este trabalho utilizará esta ferramenta para analisar o código Java do repositório.

Nas subseções abaixo serão listados mais detalhes sobre cada uma das métricas utilizadas.

4.1.1 *Issues*

A métrica *issues* é uma das métricas oferecidas pelo Sonarqube. De acordo com a documentação do (SONARCLOUD, 2023), essa métrica é dividida em três categorias:

- ***Bugs***: Erros de codificação que fazem o código-fonte não funcionar corretamente.
- **Vulnerabilidades**: Códigos que oferecem riscos a ataques mal-intencionados.
- ***Code smells***: Problemas de manutenabilidade que podem tornar o código confuso e de difícil manutenção.

Cada uma dessas categorias é subdividida em cinco criticidades:

- ***Blocker***: *Bug* com alta probabilidade de impactar o comportamento do aplicativo em produção: vazamento de memória. O código deve ser corrigido imediatamente.
- ***Critical***: Pode ser um *bug* com baixa probabilidade de impactar o comportamento do aplicativo em produção, ou uma questão que representa uma falha de segurança. O código deve ser revisado imediatamente.
- ***Major***: Falha de qualidade que pode impactar fortemente a produtividade do desenvolvedor.
- ***Minor***: Falha de qualidade que pode impactar ligeiramente a produtividade do desenvolvedor.
- **Info**: Nem um *bug* nem uma falha de qualidade, apenas uma constatação.

4.1.2 Cobertura de testes de unidade

Os testes de unidade são de extrema importância para garantir a qualidade do sistema. Aspectos como análise de valor limite, cobertura de condições dentro dos métodos e linhas executadas pelos testes são usadas para compor este percentual, que também é analisado pelo Sonarqube. Adicionalmente, este tipo de teste tem a capacidade de garantir o comportamento do sistema perante a refatorações. Por exemplo, ao verificar que uma análise do Sonarqube identificou um código com alguma *issue* que precise de alteração, o teste é capaz de garantir o comportamento anterior pois, ao ser executado, deve ter sucesso mesmo com a mudança de código realizada.

4.1.3 Trechos de código-fonte duplicados

Trechos duplicados também são um indício de que o código-fonte não tem uma separação de responsabilidades adequada. A duplicação de código-fonte pode ser traduzida como falta de centralização do código. Trechos de código-fonte duplicados oferecem maior esforço para cobertura de testes de unidades e eventuais alterações.

4.1.4 SonarCloud

O SonarCloud é um sistema *web* que executa o Sonarqube *online*, removendo a necessidade de instalação local do mesmo, além de oferecer integrações com o GitHub, podendo disparar análises baseadas em eventos ocorridos no repositório, como *pull requests* (PR) sobre uma ramificação específica. Também possui versões gratuitas e pagas, cada uma oferecendo um pacote de funcionalidades específicas. Adicionalmente, oferece diversos recursos de relatórios, onde é possível acompanhar e analisar cada um dos indicadores mencionados acima, além de diversos outros, compo é possível verificar na Figura 4.1.

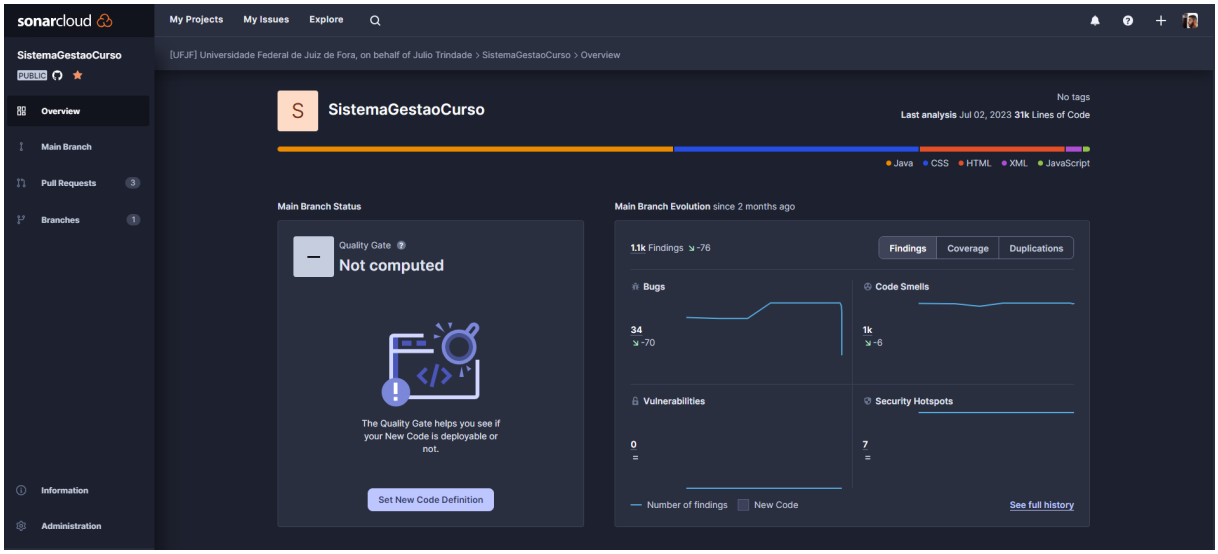

Figura 4.1: SonarCloud - Visão geral

5 Metodologia

5.1 Caracterização da pesquisa

Para este trabalho será feita uma pesquisa exploratória sobre o repositório do projeto "Sistema de Gestão de Curso" para que seja possível identificar possíveis *issues* que existam no código-fonte. A ferramenta utilizada para esta pesquisa será o *Sonarqube*. Este passo da pesquisa será denominado "passo 0" (P0).

Se identificadas *issues*, será possível prosseguir para o passo 1 (P1) da pesquisa. O relatório gerado pelo *Sonarqube* será utilizado como insumo inicial e, à partir deste momento, serão traçadas estratégias afim de apresentar melhorias contínuas do código à medida que o trabalho avança. No passo 2 (P2), serão apresentados os resultados obtidos aplicando sistematicamente o planejamento e execução descritos.

5.2 Análise inicial e obtenção de dados do P0

Para acompanharmos a evolução do trabalho, consideraremos o P0 sendo o momento em que a operação de *clone* do GitHub do projeto foi realizada, que pode ser verificada em (GITHUB, 2023). Para este trabaho, apenas levaremos em consideração as *issues* apontadas em arquivos Java (.java) e arquivos de configuração, como pom.xml e semelhantes. Não aplicaremos correções a arquivos de *Hiper Text Markup Language* (HTML), *Cascade Style Sheet* (CSS) e Javascript (JS). Caso existam *issues* nestes contextos, serão marcadas com o *status "Won't fix"*.

Issues

No momento P0, temos 104 *bugs*, 1.021 *code smells* e 0 vulnerabilidades, como podemos observar no gráfico exibido na Figura 5.1.

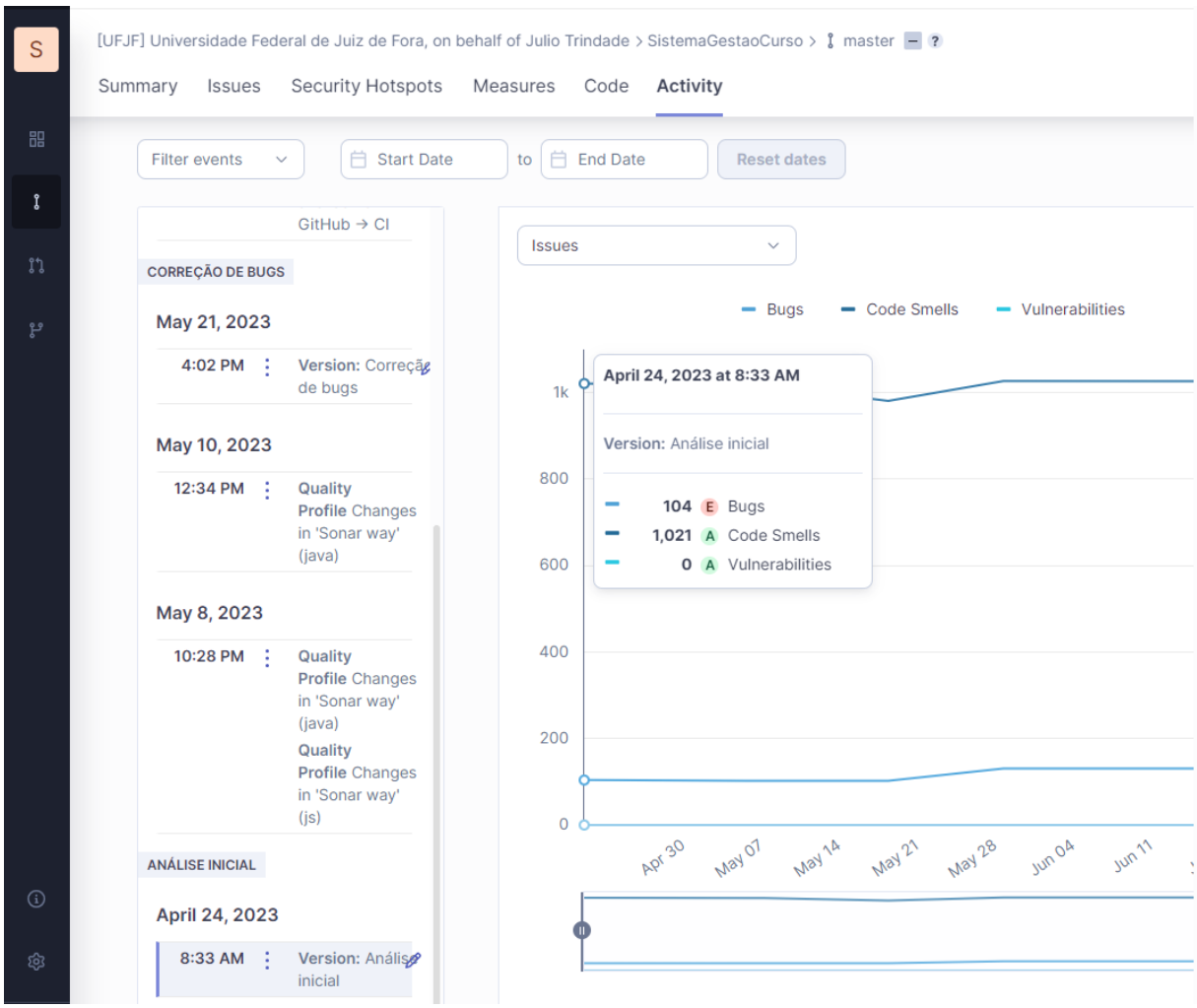

Figura 5.1: Issues encontradas após pesquisa exploratória

Cobertura de testes de unidade

No momento P0, temos 0% de cobertura de testes de unidade, como podemos observar no gráfico exibido na Figura 5.2.

Linhas de código-fonte duplicadas

No momento P0, temos 4.085 linhas duplicadas de um projeto que possui 31.162 linhas de código, o que resulta em 10,3% de duplicação de código-fonte detectados, como pode ser observado no gráfico exibido na Figura 5.3.

Como é possível observar, o momento P0 revelou que, como suposto, o projeto alvo deste trabalho contém ítens que impactam, tanto os usuários finais do sistema quanto os desenvolvedores responsáveis pela evolução e manutenção do mesmo. Com isso, é possível avançar para o momento P1, onde será estabelecida a metodologia de trabalho

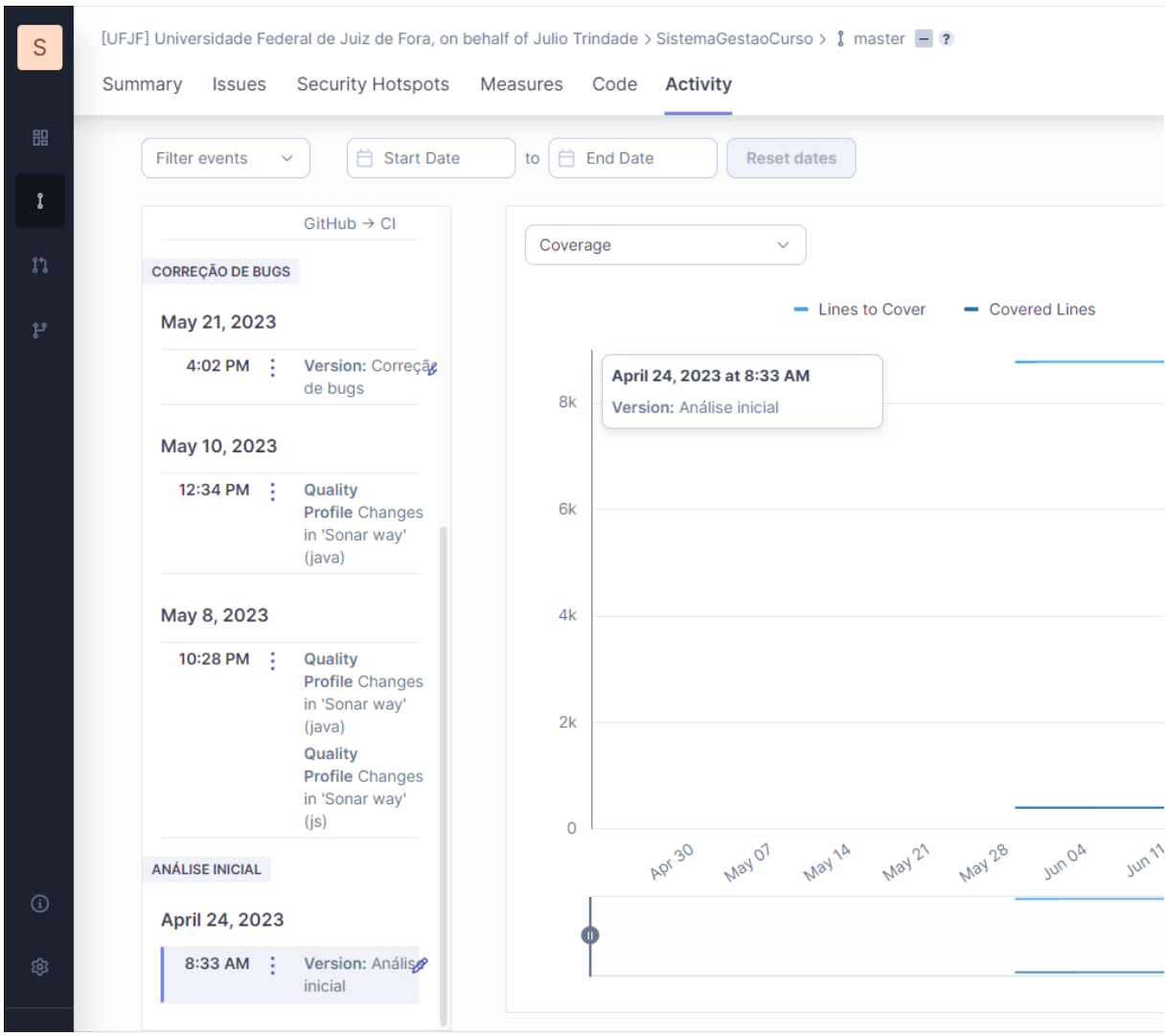

Figura 5.2: Índice de cobertura de testes de unidade após análise exploratória

para melhoria dos índices.

5.2.1 Planejamento do P1

Baseado nos resultados da pesquisa exploratória realizada,será criado um mecanismo de CI para executar verificações sistemáticas e recorrentes baseada em eventos disparados pelos desenvolvedores em sua rotina de trabalho, como *commits* e *pull requests* para que seja possível chegar ao objetivo de não permitir que uma falha ou *smell* permanaça no sistema por muito tempo sem ser notada (CARDOSO MARCELO DE CASTRO E BARBOSA, 2018).

O *Sonarcloud*, como ferramenta de análise, oferece uma tela de relatório que lista todas as *issues* encontradas, separando-as por tipo, severidade, *status* e diversos outros.

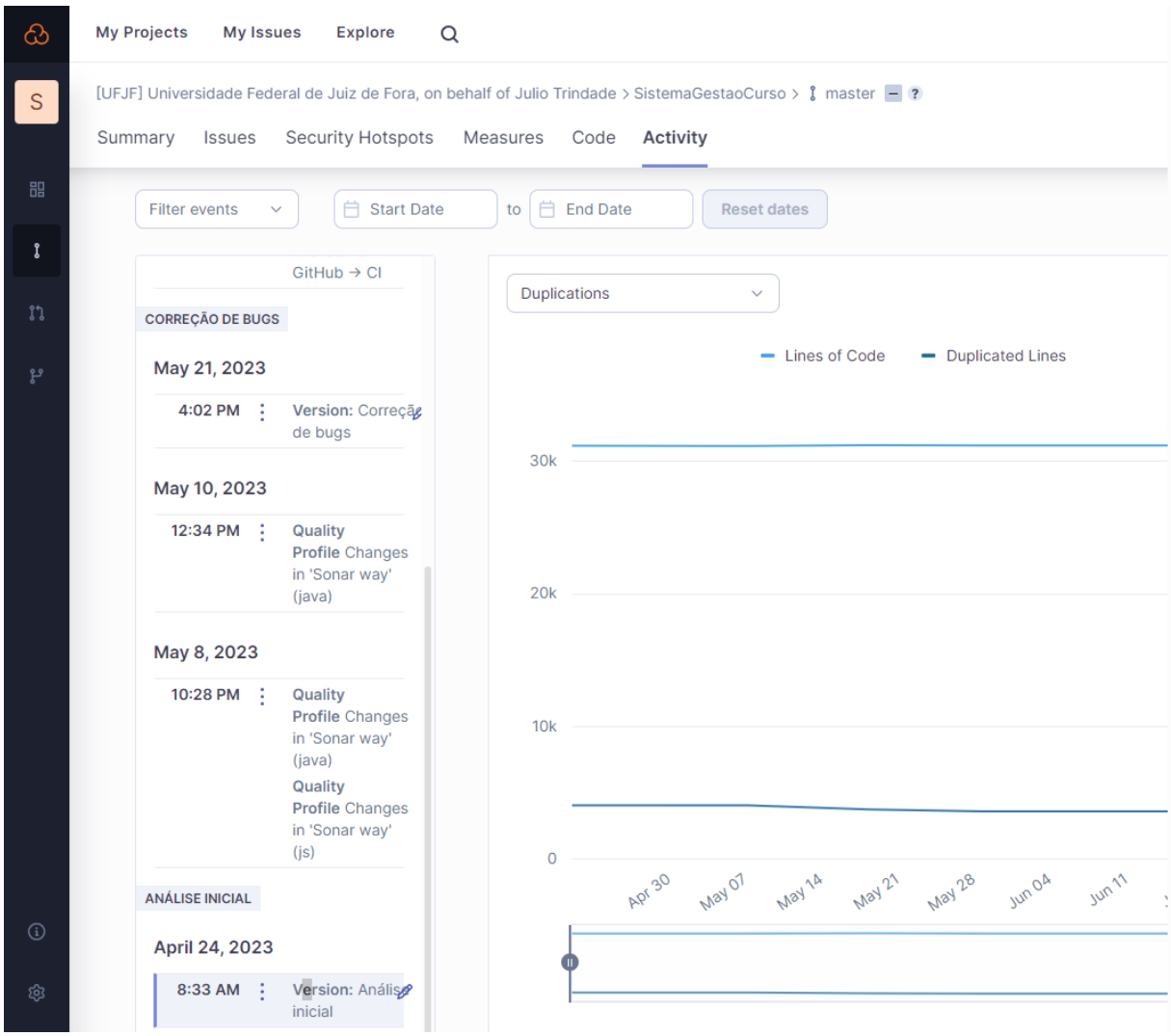

Figura 5.3: Índice de duplicação de código após análise exploratória

Como foco deste trabalho, será utilizada a seguinte metodologia de trabalho: Para cada tipo de *issue*, seguindo a enumeração da listagem feita, serão aplicadas correções dos ítens de severidade *blocker*, *critical* e *major*.

Como passo inicial da execução e para ilustrar a metodologia adotada, abaixo, na Figura 5.4, é possível visualizar o primeiro filtro utilizado.

É possível verificar que, para o tipo de *bug*, não existem severidades *blocker* e *critical*. Com isso, a próxima severidade a ser atacada será a *major*. Neste momento do trabalho serão necessárias alterações no código-fonte. Como o sistema não possui testes de unidade que validem o comportamento do código, estes serão criados visando garantir que após a correção o comportamento do sistema se manterá.

Do mesmo modo, durante a análise, caso sejam identificados pontos de código duplicado, estes serão unificados. A fim de ordenar o trabalho realizado, as *issues* de

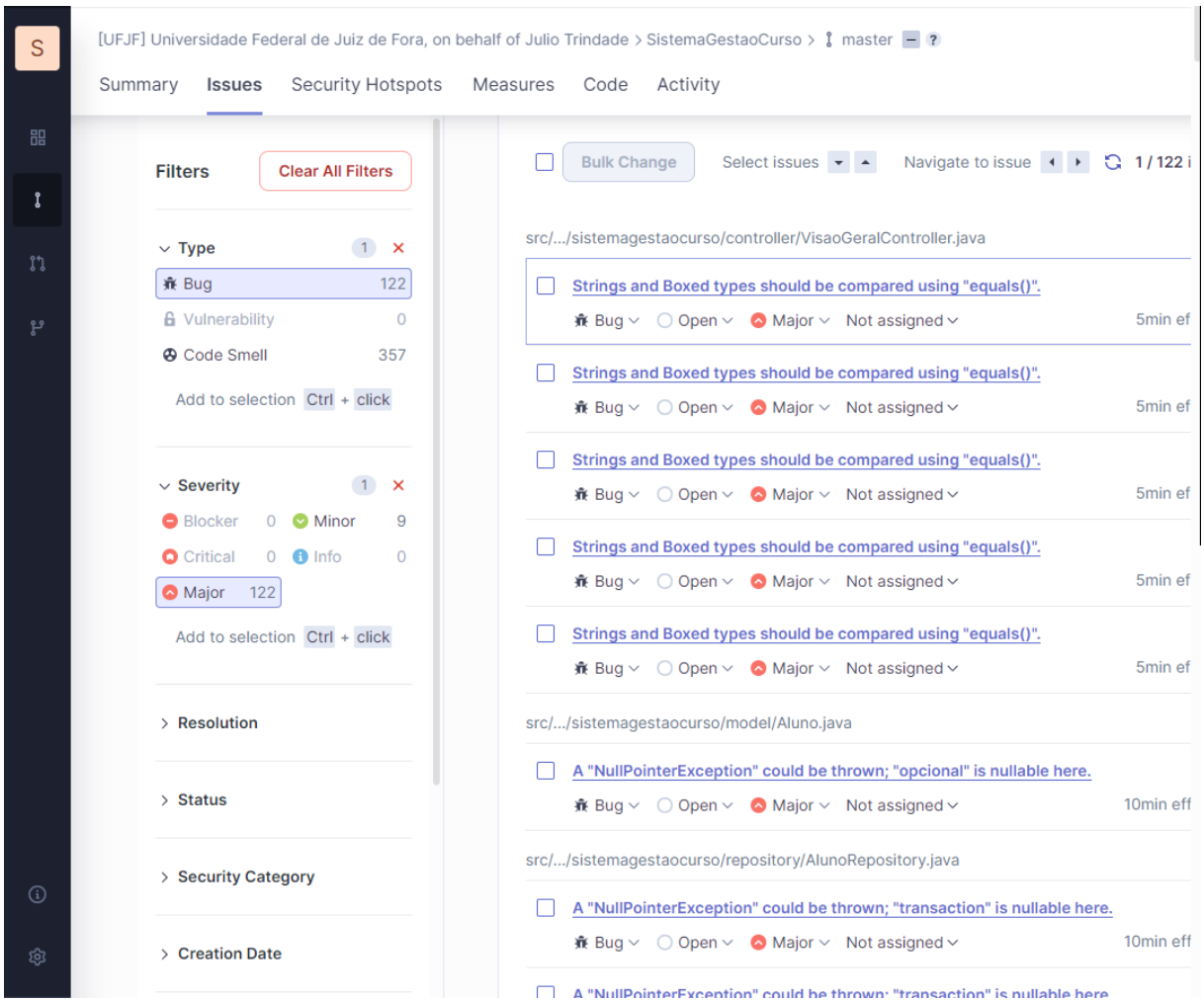

Figura 5.4: Filtro de bugs por severidade *major*

mesma severidade serão resolvidas por arquivo e para cada tipo de *issue* em cada tipo de severidade, uma ramificação (*branch*) será criada no GitHub, seguindo a nomenclatura "sonarqube/<*issue type*>/<severidade>".

Á medida que os ítens forem sendo ajustados, *pull requests* serão gerados da ramificação criada para a ramificação *master*, que é a ramificação padrão do projeto, onde são executadas as análises de forma automática pelo CI construído no P0 do projeto.

Ao fim das iterações, uma *tag* será associada para visualização facilitada da melhoria obtida.

Dada a metodologia, espera-se que a correção sistemática de *issues* contribua diretamente com os outros índices mencionados anteriormente.

5.3 Execução

Nas subseções baixo, as etapas de execução do trabalho de diminuição do número de *bugs* e *code smells* serão listadas. Dentro de cada uma das subseções, os detalhes de cada uma das etapas serão mencionados.

5.3.1 *Bugs*

Inicialmente foram apontados cento e vinte e dois (122) itens para esta categoria. Após filtragem, foi observado que setenta e nove (79) eram relacionados com arquivos HTML, CSS e JS. Estes ítens foram removidos da listagem. Com isso, quarenta e três (43) ítens restaram na listagem. Seguindo a metodologia descrita anteriormente, a ramificação "sonarqube/*bugs*/*major*" foi criada para resolver os problemas relativos à código java que foram apontados pelo *Sonarqube*, e estes problemas serão resolvidos por arquivos.

VisaoGeralController.java

O *Sonarqube* analisou a classe VisaoGeralController.java reportando os *bugs* mostrados na Figura 5.5.

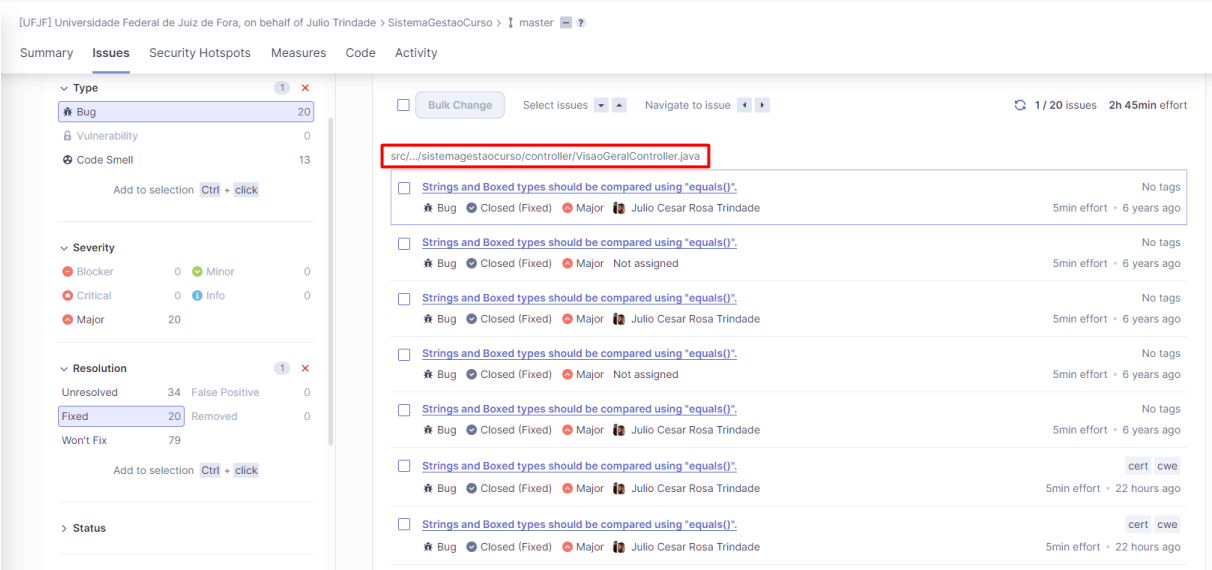

Figura 5.5: *Bugs* de severidade *major* reportados no arquivo VisaoGeralController.java

Ao acessar o *hiperlink* referente à *issue*, é possível verificar detalhes sobre o que é este item, porque o mesmo foi reportado, exemplos de código compatíveis e não compa-

tíveis com a solução. Neste caso, o *Sonarqube* reportou que comparação de tipos *String* não deve ser feita usando o operador "==", mas sim usando o método *String.equals*().

Durante a análise da correção, foi identificado que para acessar este item através de testes de unidade, seria necessário invocar o método VisaoGeralController.gerarDados() e construir os cenários necessários para cobrir a sequência lógica até este ponto, assim como seus valores de decisão. Adicionalmente, foi verificado que seu tamanho era de cento e noventa e seis (196) linhas.

A quantidade de linhas deste método sugere que, possivelmente, serão encontrados altos índices de complexidades cognitivas, ciclomáticas e esforço de entendimento, teste e manutenção, o que dificultaria a geração da cobertura de testes. Foi possível verificar esta suspeita acessando a seção de complexidade, no próprio *Sonarcloud*, como visto na Figura 5.6.

Ao analisar mais cuidadosamente o método, é possível notar que este desempenha uma série de operações de cálculo de grade, avaliação de currículo do aluno, dentre outras. Empiricamente, é possível supor que esta classe possui mais responsabilidades do que deveria possuir.

O S, do acrônimo SOLID, denota o princípio da responsabilidade única vai diretamente de encontro ao método VisaoGeralController.gerarDados(), que possui diversas responsabilidades.

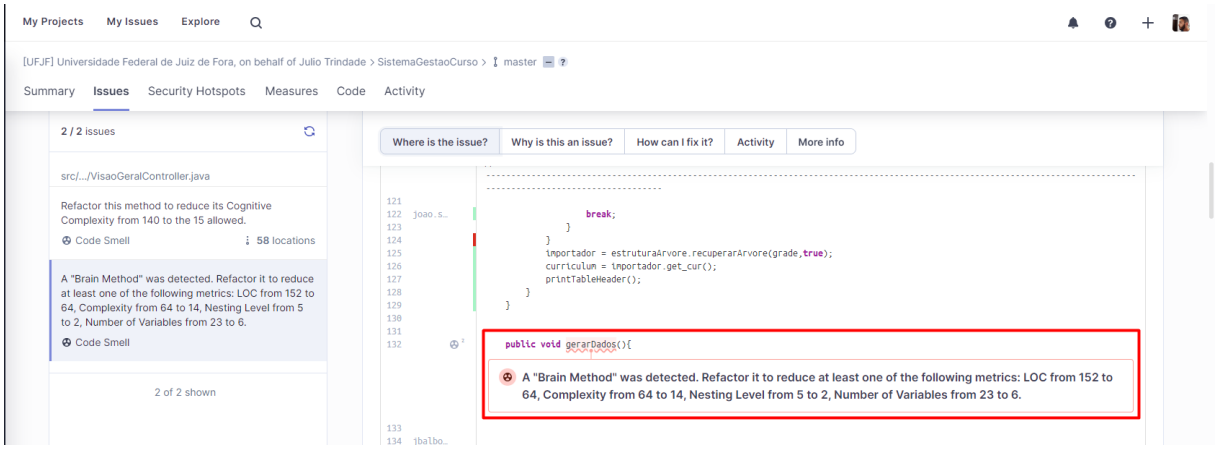

Figura 5.6: Complexidade do método VisaoGeralController.gerarDados()

Para construir o teste de unidade que cobre o ponto a ser alterado, foram necessárias mudanças estruturas na classe e a criação de uma classe de objetos *fake*.

O Mockito foi utilizado para criar um provedor *mock* e, definimos que este provedor, ao ser acessado pela classe VisaoGeralController, iria fornecer uma instância do objeto EstruturaArvore com suas propriedades alteradas para atingir o fragmento de código a ser testado. Neste caso tornou-se necessário criar uma classe que fosse capaz de reproduzir o relacionamento complexo entre os objetos operados pelo método que está sendo testado.

A classe criada foi a Mocks.java, e se encontra no diretório de testes de unidade da aplicação. Esta classe foi excluída da análise de código-fonte pois a mesma não é uma classe de teste, que já é automaticamente excluída, e também não oferece funcionalidades à aplicação.

No âmbito das mudanças estruturais, foi criado um provedor para a classe EstruturaArvore, que era instanciada diretamente em um dos métodos de inicialização utilizando o padrão *Singleton*. Este provedor passou a ser injetado como dependência. Com isso, se torna viável fazer o provedor retornar um objeto *fake* para testes. Com a junção dessas duas técnicas, foi possível cobrir as condições necessárias para efetuar a correção do item.

A Figura 5.7 mostra o trecho do código antes e depois da alteração. As linhas em cinza demonstram o estado do código antes das alterações e as linhas em verde demonstram o resultado depois das alterações.

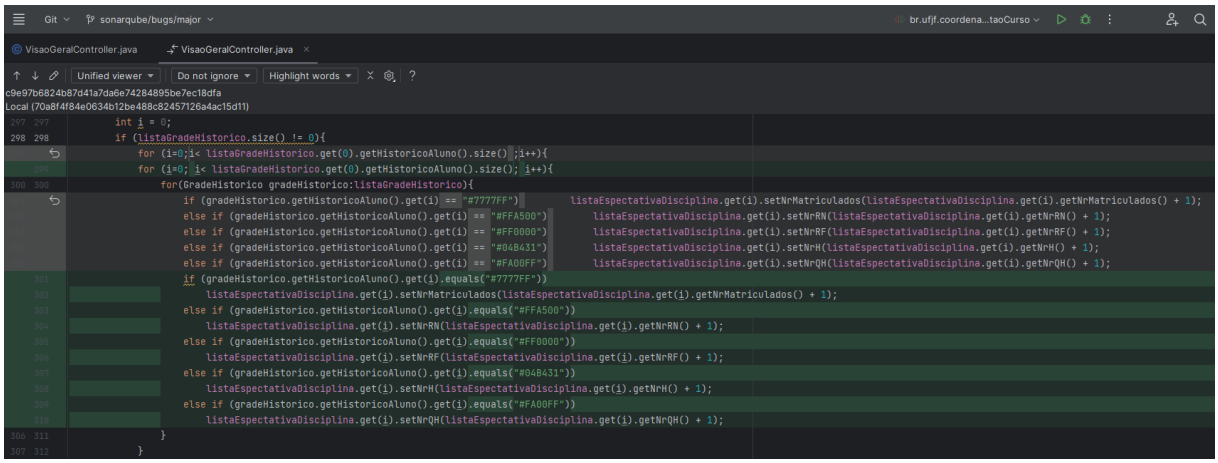

Figura 5.7: Trecho de código reportado pelo *Sonarqube* no arquivo VisaoGeralController.java

Ao efetuar a análise novamente, os ítens reportados foram movidos automaticamente para a seção *fixed* do *Sonarqube*, porém, a métrica de cobertura de testes de

unidade teve um comportamento não esperado. Com a cobertura de testes de unidade do método VisaoGeralController.gerarDados() até o ponto necessário para o item reportado, o índice geral de cobertura de código da aplicação subiu para 9%, como pode ser visto na Figura 5.8.

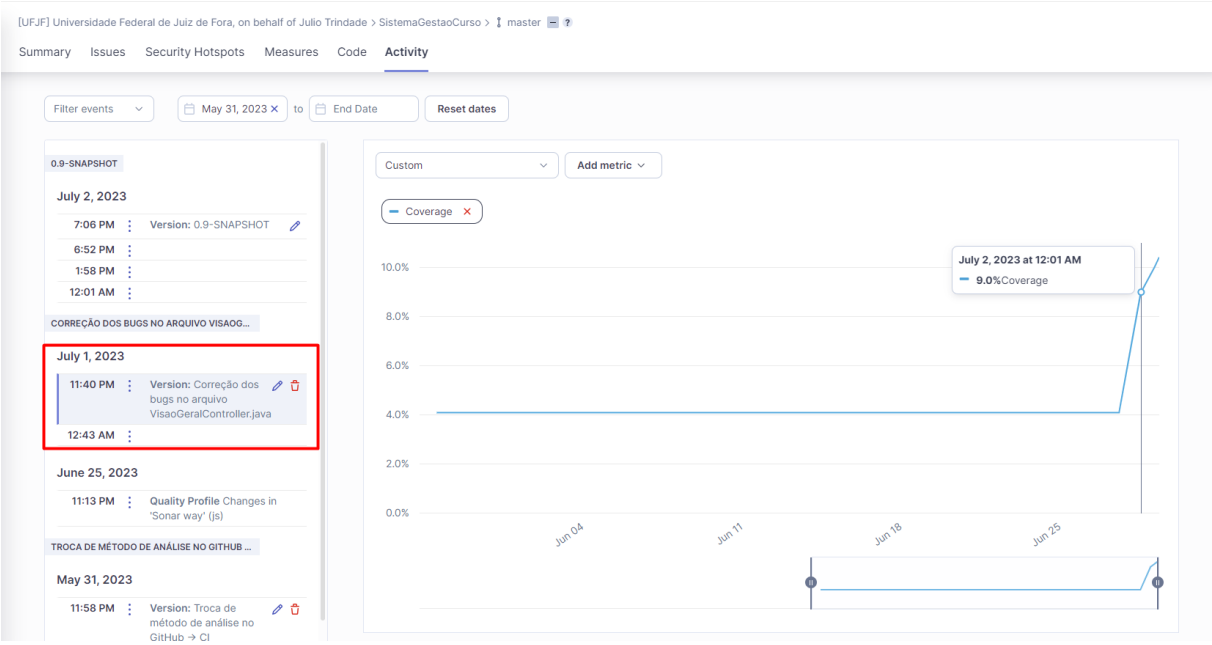

Figura 5.8: Cobertura de código após testes de unidade para VisaoGeralController.gerarDados()

.

É possível inferir que um método que é capaz de atingir 9% de todas as linhas de um projeto de software utiliza funções de diversas outras classes. Quando as propriedades da classe VisaoGeralController são observadas, é verificado que esta possui diversas propriedades que referenciam outras classes do sistema e também faz uso da classe EstruturaArvore, que também interage com várias outras entidades. É possível definir esta interdependência como alta conectividade de classes. Ferreira, Bigonha e Bigonha (2008) definem este aspecto como sendo um dos principais para definição da complexidade em se manter e alterar um sistema. Esta alta conectividade significa que, ao alterar uma das classes conectadas, existe alta probabilidade de mudanças serem propagadas por diversos outros pontos, acarretando em comportamentos inesperados.

Aluno.java

A classe Aluno.java foi reportada contendo um *bug* de possível disparo de exeção de referencia nula (*NullReferenceException*). Esta situação pode ocorrer quando, em algum caminho lógico do algoritmo, temos uma referência nula que pode tentar ser acessada, como pode ser verificado na Figura 5.9.

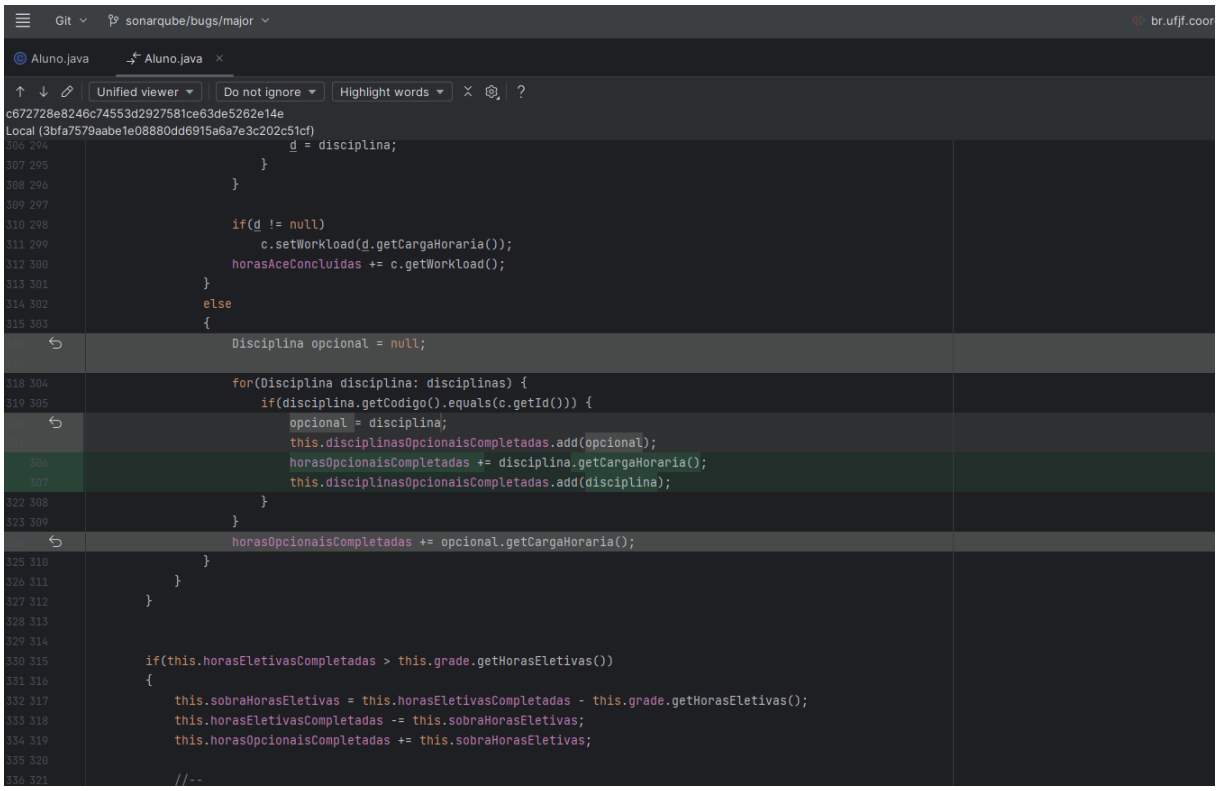

Figura 5.9: recho de código reportado pelo *Sonarqube* no arquivo Aluno.java

Utilizando a mesma metodologia utilizada anteriormente, foi identificado qual método era necessário acessar para atingir o trecho de código reportado. Ao analisar o método Aluno.calculaHorasCompletadas(), foi identificado um cenário semelhante ao método anteriormente corrigido. Este método possui noventa e seis linhas (96), muitas estruturas de repetição, condicionais e tem bastante interação com outras classes do sistema (acoplamento alto e coesão baixa). Convenientemente, foi utilizado o provedor de EstruturaArvore e da classe de *mocks* criados anteriormente para ganhar celeridade no processo de cobertura de testes e ajuste do reporte.

Ao fazer o uso destes artificios, a cobertura de testes de unidade subiu de 9% para 10%. Do mesmo modo, este método apresenta índices de complexidade cognitiva e ciclomática, 62 e 23, respectivamente, também indicando possível baixa coesão e alto

acoplamento, comprometendo a testabilidade, reusabilidade e manutenção deste.

AlunoRepository.java

Para a classe AlunoRepository.java, também foi observada uma *issue* relacionada a *NullReferenceException*, porém dessa vez em um com poucas linhas, baixa complexidade cognitiva e ciclomática.

Após cobertura de testes, foi verificado que existia duplicação de código para este trecho. Ao analisar, foi constatado que todas as classes <Prefixo>Repository.java possuíam exatamente o mesmo método. Utilizando técnicas de OO foi possível unificar o trecho utilizando uma classe abstrata, que foi chamada de BaseRepository.java. Nela, foi utilizado o recurso da linguagem Java chamado *Generics* para conseguir torná-la flexível o suficiente para lidar com os diferentes tipos de entidades persistidas. As alterações podem ser vistas nas figuras Figura 5.10 e Figura 5.11.

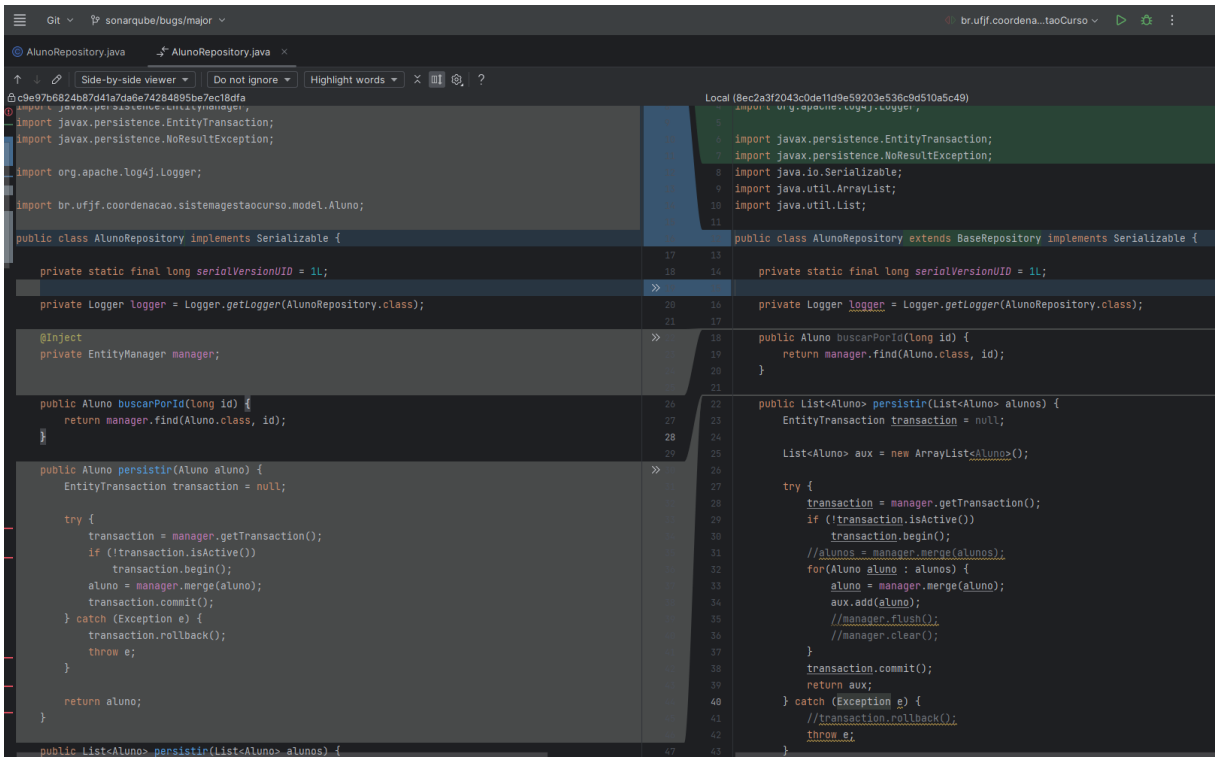

Figura 5.10: Trecho de código reportado pelo *Sonarqube* no arquivo AlunoRepository.java

Após realizadas estas alterações, o método AlunoRepository.persistir(Aluno aluno) pôde ser substituído pela implementação <T> T BaseRepository.persistir(T entity). Aproveitando esta estrutura, o mesmo foi feito para outras classes *Repository* que continham a duplicação de código.

A cobertura de testes também se beneficiou desta técnica, pois foi possível criar uma classe de testes que testa todos os repositórios que herdam da classe BaseRepository.java. Com esta mudança, além da correção dos ítens reportados para cada *Repository* com a mesma *issue*, obtivemos uma melhora no percentual de duplicação de código, que saiu de 9,1% para 8,9%, segundo relatório do *Sonarcloud*. A complexidade de testes deste método também não foi alta, ratificando que existe uma relação direta entre esforço de testes e complexidades cognitivas e ciclomáticas.

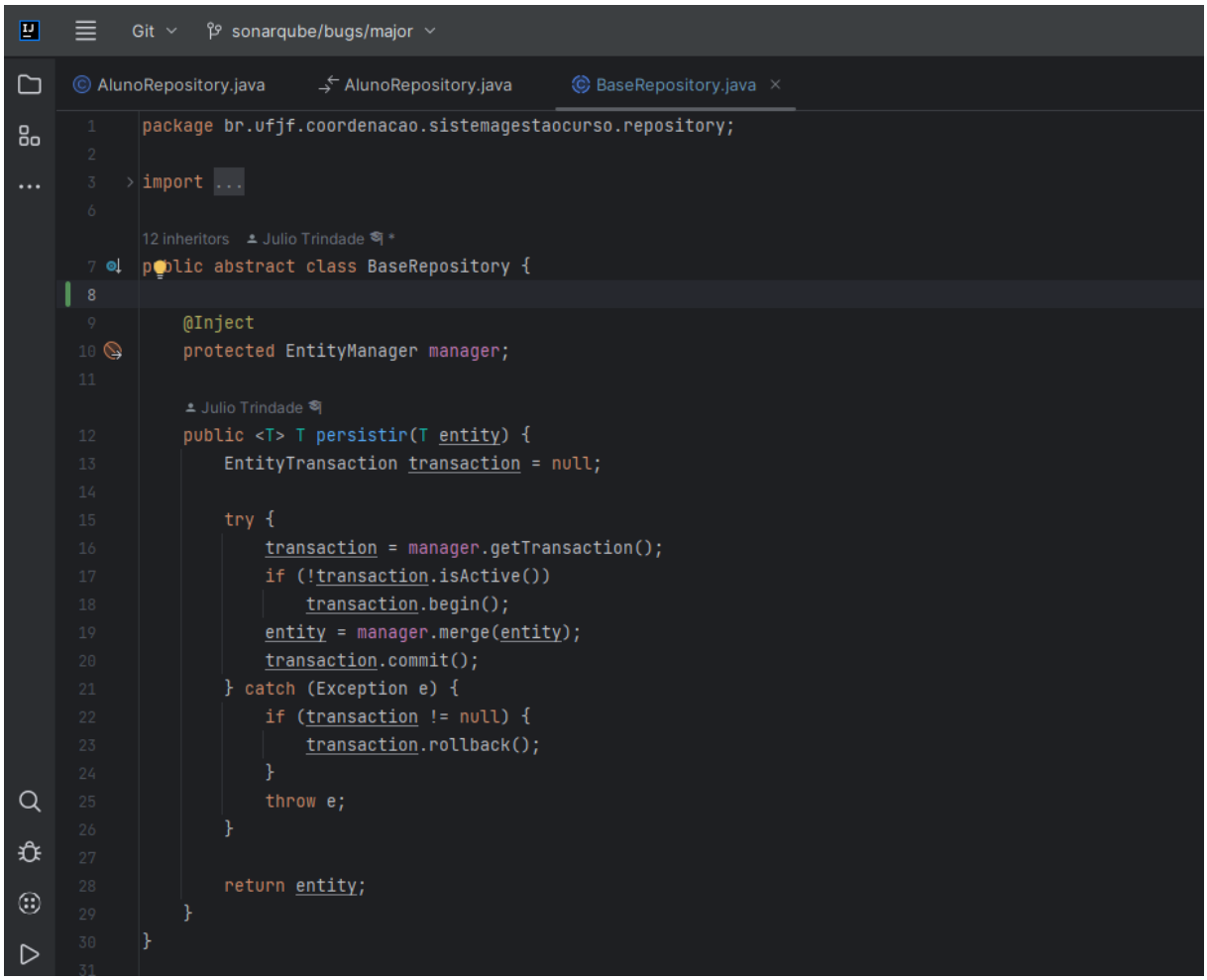

Figura 5.11: Classe abstrata BaseRepository.java

6 Resultados

Os resultados deste trabalho podem ser divididos entre "documentação e repositório", que dizem respeito à todos os passos executados para execução do projeto, posteriormente documentados no próprio repositório e "Código-fonte e métricas", descrevendo as alterações de código-fonte e os benefícios causados por elas, em análises recorrentes do Sonarqube.

6.1 Documentação e repositório

Durante o trabalho realizado entre P0 e P1, diversos desafios relacionados à configurações do sistema foram encontrados. Estes desafios ocorreram, em sua grande maioria, pela ausência de uma documentação clara, que fornecesse dados sobre as dependências do projeto, versões de *frameworks*, versões de Java *development kit* a serem usadas, dentre outras coisas. Conforme proposto, uma documentação foi criada para centralizar estas informações e foi anexada ao repositório do projeto no GitHub, como pode ser visto na Figura 6.1.

Adicionalmente, insígnias sobre a qualidade do código foram inclusas para facilitar a visualização por parte dos próximos desenvolvedores. A documentação também relaciona documentações relativas a processos de testes de unidade, convenções sobre fluxo Git e *pipelines* de CI criados para análise contínua.

Ainda referente a CI, o arquivo ".github/workflows/maven.yml" foi criado para que o repositório seja capaz de guardar a definição do *pipeline* "(Maven) Build, test and analyze", que é responsável por efetuar a compilação, executar os testes de unidade e análises do Sonarqube, publicando-as no SonarCloud. Esta é uma forma de manter análise contínua sobre o código do projeto. A Figura 6.2 mostra como o GitHub exibe o resultado da análise feita pelo SonarCloud.

Ainda no GitHub, foi criado um modelo de texto para *pull requests*, de modo a incentivar os desenvolvedores a preencherem, de forma clara e padronizada, o sumário de suas alterações, como pode ser visto na Figura 6.3. Esta alteração visa motivar os

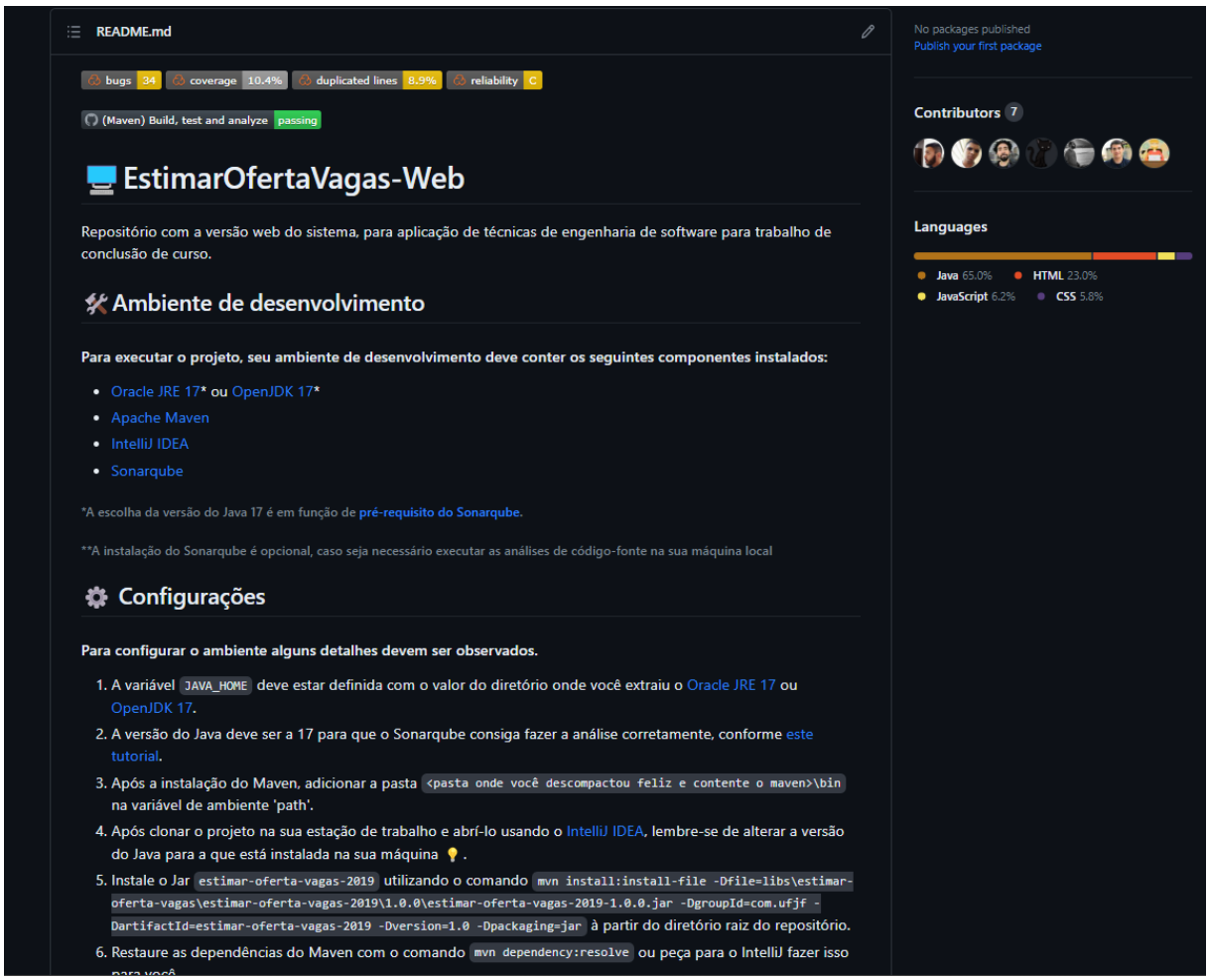

Figura 6.1: Repositório do projeto - Readme.md

desenvolvedores a fornecerem mais riqueza de detalhes e permitir que os revisores consigam ser mais assertivos nas suas análises para integração de novo código.

Além disso, o *pull request* executa o *pipeline* de CI todas as vezes que é criado e alterado. Isso torna o processo de verificação automatizado, criando assim rastreabilidade de surgimento de novos *bugs* e *code smells*.

6.2 Código-fonte e métricas

Como resultado da aplicação da métodologia, os seguintes resultados foram encontrados. Os resultados seguem os marcos P0 e P1, definidos anteriormente.

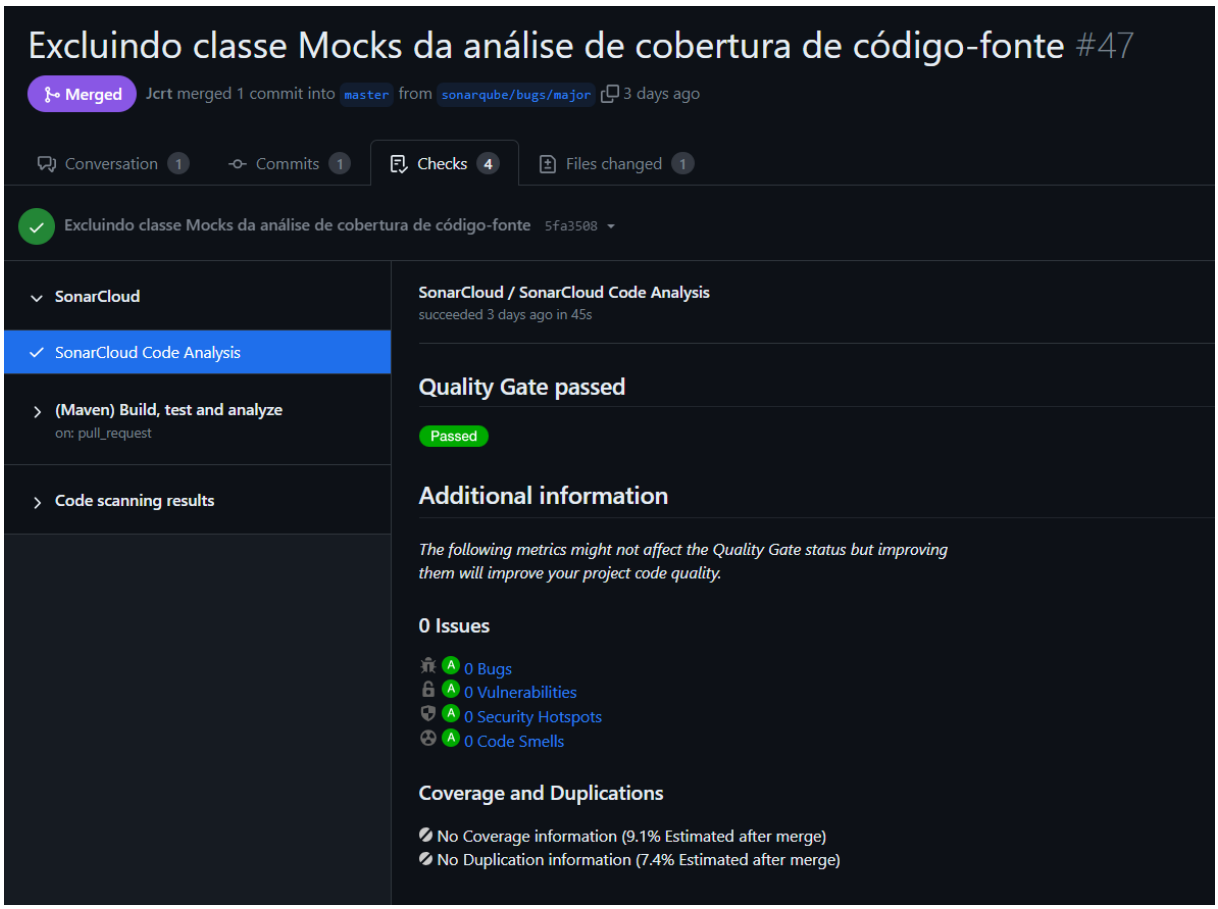

Figura 6.2: Repositório do projeto - Resultado da análise feita pelo CI

6.2.1 *Bugs*

Os *bugs* foram os primeiros ítens da análise afetados pelo trabalho. Durante a análise, foram observados, em sua maioria, itens de HTML, CSS e JS, que foram excluídos dos ajustes. Após esta exclusão, restaram quarenta e três itens. Destes, foram ajustados nove. Não foi possível concluir a lista completa pelo alto custo para geração de código de teste de unidade. Os números podem ser verificados na Tabela 6.1.

Bugs por criticidade	T0	T1	% de melhoria
Blocker	0	0	0%
Critical	0	0	0%
Major	43	34	21%
Não ajustados (HTML, CSS, JS)	79	0	-

Tabela 6.1: Bugs por criticidade

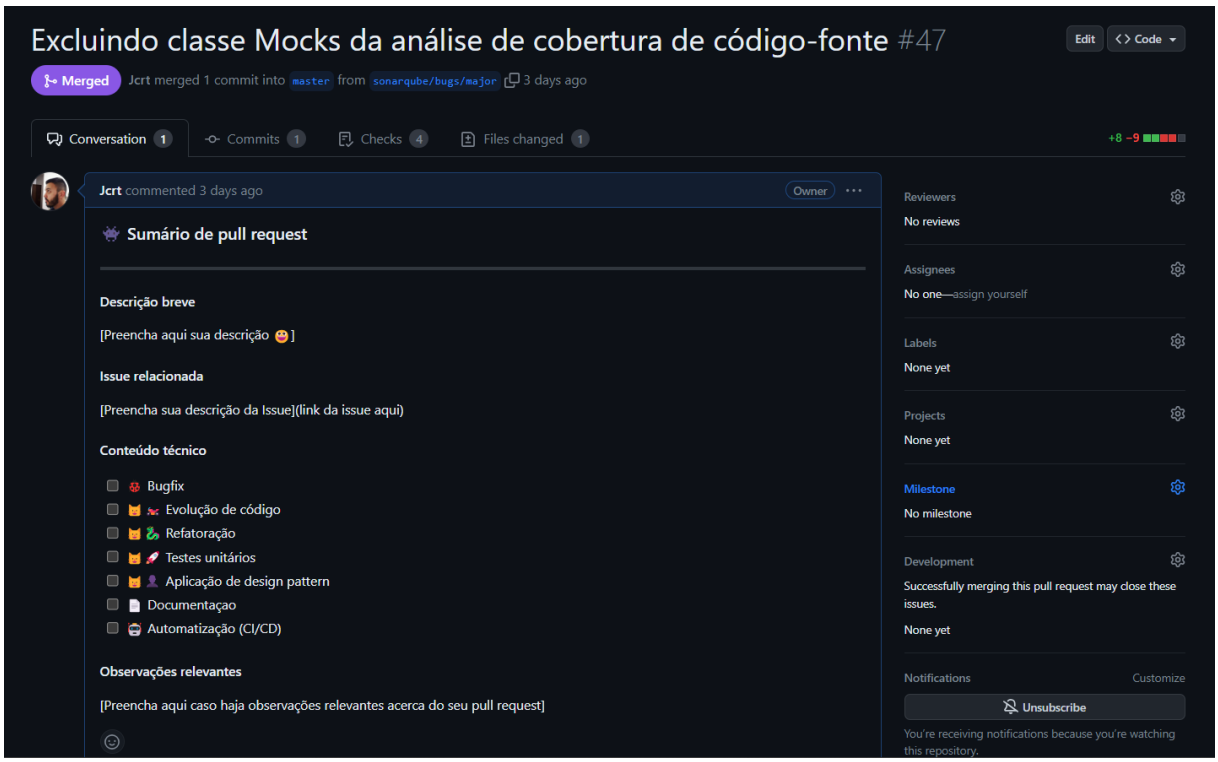

Figura 6.3: Repositório do projeto - template de pull request

6.2.2 *Code smells*

Os *Code smells*, entre T0 e T1, não foram atacados diretamente, pois houve um alto esforço para *bugs*. Porém, houve melhora de forma indireta. A correção de um *code smell blocker* aconteceu no processo de configuração do sistema, pois era relacionado à forma como a dependência "com.ufjf.estimar-oferta-vagas-2019" era definida no arquivo pom.xml.

Durante a construção do *pipeline* de CI, foi necessário ajustá-la. Este tipo de correção não envolve testes de unidade e não é relativa a arquivos Java, por isso, o sucesso na compilação do projeto já é capaz de validar sua corretude. Outros itens, considerados *critical*, foram ajustados ao passo que modificações para correção de *bugs* foram realizadas. Os números podem ser verificados na Tabela 6.2.

Code smells por criticidade	T0	T1	% de melhoria
Blocker	5	4	20%
Critical	246	242	1%
Major	357	357	0%

Tabela 6.2: *Code smells* por criticidade

6.2.3 Outras métricas

É possível citar como dados relevantes para este trabalho as métricas de cobertura de código por testes de unidade e de duplicação de código-fonte. A primeira apresentou melhorias significativas, devido à aplicação de testes para alteração da classe VisaoGeralController.java e Aluno.java. A duplicação de código apresentou redução do seu percentual devido à utilização de conceitos de orientação a objetos para unificar códigos que eram literalmente iguais e com mesma semântica. Os números podem verificados na Tabela 6.3.

Outras métricas	T0	T1	% de melhoria
Percentual de cobertura de código por testes de unidade	0	10,4	10,4%
Percentual de duplicação de código-fonte	10,3	8,9	13,59%

Tabela 6.3: Outras métricas

7 Conclusão

Após as análises realizadas por este trabalho sobre o sistema "Sistema de Gestão de Curso", foi possível constatar que projetos de impacto administrativo, gerencial e operacional são de extrema importância para os departamentos de uma corporação, o que justifica o esforço em construí-los. Podemos notar também que, ao tempo que a necessidade destes surge, nem sempre é possível realizar um planejamento adequado para execução e gestão do projeto.

Estes fatores, juntamente com a falta de experiência dos desenvolvedores responsáveis pela evolução e manutenção, podem tornar muito onerosa a operação de desenvolvimento e manutenção a longo prazo pela crescente adição descoordenada de complexidade e acoplamentos entre as unidades do sistema.

A falta de implementação de testes de unidade torna qualquer alteração no sistema arriscada, pois não se têm estruturas que garantam os comportamentos após aplicação de correções de código-fonte.

Durante a evolução do projeto não houve construção de documentações efetivas para disseminação de conhecimento e diminuição da curva de aprendizado do sistema.

Adicionalmente, não existem ambientes de desenvolvimento e testes para manutenção, o que torna a etapa de configuração e execução em ambientes de desenvolvimento consideravelmente mais onerosa.

Embora o projeto esteja dentro de um repositório de código-fonte, não foi detectado nenhum tipo de fluxo de entrega de novas implementações.

Empiricamente, foi observado que as estimativas de tempo gasto para ajustes das *issues* e *code smells* feitas pelo Sonarqube foram subestimadas, pois não levam em conta a construção de testes de unidade para cobrir o ponto de código a ser alterado e nem mesmo as complexidades ciclomáticas e cognitivas do contexto em que este código está localizado.

O aumento da cobertura de testes de unidade deve ser considerado algo prioritário, pois esta técnica abre possibilidade para que técnicas de desacoplamento e centrali-

zação de código-fonte sejam aplicadas com segurança.

Com a construção de um fluxo de entrega de código-fonte aliado à CI, foi possível gerar automaticamente o acompanhamento das métricas ao passo que o novo código era integrado. Isso permitiu que os dados deste trabalho fossem obtidos com pouco esforço. Adicionalmente, eliminou a etapa de configuração do Sonarqube localmente, diminuindo ainda mais o tempo de configuração da estação de trabalho local.

A constatação de alto acoplamento e baixa separação de responsabilidades presente no código-fonte tornou muito oneroso o processo de correção de *bugs* e *issues*, conforme previsto na literatura.

7.1 Trabalhos futuros

No intuito de indicar direções para que a evolução e manutenção do sistema "Sistema de gestão de curso" possa ser simplificada e melhorada, alguns trabalhos futuros podem ser sugeridos.

- Utilização de sistemas de gestão de incidentes
- Construção de ambiente de desenvolvimento
- Aplicação de técnicas de redução de acoplamento
- Aplicação de programação multi camadas
- Criação de *templates* para novos projetos

Bibliografia

ALFAYEZ, R.; BEHNAMGHADER, P.; SRISOPHA, K.; BOEHM, B. An exploratory study on the influence of developers in technical debt. In: *Proceedings of the 2018 International Conference on Technical Debt*. New York, NY, USA: Association for Computing Machinery, 2018. (TechDebt '18), p. 1–10. ISBN 9781450357135. Disponível em: <https://doi.org/10.1145/3194164.3194165>.

ARAKAKI, J.; FERREIRA, B. N. Aplicação de métricas de manutebilidade na refatoração de softwares. In: *14th CONTECSI - International Conference on Information Systems and Technology Management*. [s.n.], 2017. Disponível em: <https://www.tecsi.org/contecsi/index.php/contecsi/14CONTECSI/paper/viewPaper/4703>.

ASSUNçãO, E. dos S. *Investigando a incidência decode smells em métodos de padrões de projeto*. Dissertação (mathesis) — Universidade Federal da Bahia, sep 2021. Disponível em: <https://repositorio.ufba.br/bitstream/ri/34648/1/PGCOMP-2021-DissertaÃğÃčo_Mestrado-Ederson_dos_Santos_AssunÃğÃčo.pdf>.

BEAL, A. Introdução à gestão da tecnologia da informação. *Vydia Tecnologia*, maio 2001. Acessado em 07-07-2023.

BENTO, G. J. T. R. *Refatoração do jogo Bicho UFC Rampage usando SOLID e padrões de projeto*. Dissertação (techreport) — Universidade Federal do Ceará, 2020. Disponível em: <http://www.repositorio.ufc.br/handle/riufc/55596>.

CARDOSO MARCELO DE CASTRO E BARBOSA, H. A. B. *Análise das vantagens do uso de ci/cd nodesenvolvimento de um projeto java web*. Dissertação (techreport) — Centro Universitário de Anápolis, fev. 2018. Disponível em: <http://repositorio.aee.edu.br/jspui/handle/aee/1097>.

FERREIRA, K. A. M.; BIGONHA, M. A. S.; BIGONHA, R. S. Reestruturação de software dirigida por conectividade para redução de custo de manutenção. *Revista de Informática Teórica e Aplicada*, v. 15, n. 2, p. 155–180, Dec. 2008. Disponível em: <https://seer.ufrgs.br/index.php/rita/article/view/rita_v15_n2_p155-180>.

FOWLER, M. *Inversion of control*. 2005. Acessado em: 04 de Julho de 2023. Disponível em: <https://martinfowler.com/bliki/InversionOfControl.html>.

FOWLER, M. *Continuous Integration*. 2006. Acessado em: 05 de Julho de 2023. Disponível em: <https://martinfowler.com/articles/continuousIntegration.html>.

FRITOLA, R. G.; SANTANDER, V. F. A. *Documentando requisitos de sistemas legados: um estudo decaso utilizando técnicas da engenharia de requisitos orientada a objetivos*. Dissertação (techreport) — Universidade Estadual do Oeste do Paraná, 2021.

GITHUB. *GitHub / Commit*. 2023. Acessado em: 30 de Junho de 2023. Disponível em: <https://github.com/Jcrt/dcc127-estimar-oferta-vagas/commit/19daecbc61b6f15fb26af6cebc51633b8a4d729c>.

GONçALVES, O. L. M. B. P. PrincÍpio da inversÃo de dependÊncia na qualidade de software: Aplicação da injeção de dependência no desenvolvimento de software. *Revista Interface Tecnológica - FATEC - Vol. 19*, 2022. Disponível em: <https://revista.fatectq.edu.br/interfacetecnologica/article/view/1362/746>.

JESUS, K. F. D. *Os mandamentos da programação modular em Java*. Dissertação (techreport) — Universidade Federal de Minas Gerais, 2016. Disponível em: <https://repositorio.ufmg.br/bitstream/1843/ESBF-A9CNVP/1/monografia___kenia_final.pdf>.

LAURINDO, F. J. B.; SHIMIZU, T.; CARVALHO, M. M. de; JR., R. R. O papel da tecnologia dainformação (ti) na estratégiadas organizações. *Revista Gestão e Produção*, v. 8, n. 2, p. 160–179, aug 2001. Disponível em: <https://www.scielo.br/j/gp/a/vt5SZnMwqNVyxFnkvJnLXCH/?format=pdf&lang=pt>.

MAGALHãES, N. M.; JUNIOR, H. de S. C.; ARAúJO, M. A. P. Melhoria da qualidade de software através da eliminação da complexidade desnecessária em código fonte. *Revista Multiverso*, v. 3, 2018.

MEIRELLES, P. R. M. *Monitoramento de métricas de código-fonte em projetos de software livre*. Dissertação (mathesis) — Universidade de São Paulo, maio 2013. Disponível em: <https://www.teses.usp.br/teses/disponiveis/45/45134/tde-27082013-090242/en.php>.

NETO, J. G. d. R. *Entendendo a relação entre integração contínua e cobertura de testes: um estudo empírico*. Dissertação (mathesis) — Universidade Federal do Rio Grande do Norte, aug 2021. Disponível em: <https://repositorio.ufrn.br/handle/123456789/47092>.

OLIVEIRA, R.; DARCE Álvaro. Como o teste de software pode ajudar o desenvolvimento. *Revista Intertemas*, v. 9, n. 9, 2013.

S.A, S. *SonarSource | Sonarqube*. 2023. Acessado em: 30 de Junho de 2023. Disponível em: <https://www.sonarsource.com/products/sonarqube/?gads_campaign=SQ-Mroi-PMax&gads_ad_group=Global&gads_keyword=&gclid=Cj0KCQjwzdOlBhCNARIsAPMwjbwFy5LBq19IaYh3hh1h1s_wuunUGEXIHfvugZje51omG8OJ3UtKURgaAsmfEALw_wcB>.

SANTOS GUILHERME VILATORO E LOPES, L. F. B. Introdução a boas práticas de programação com java, usando padrões de projeto. *Revista de Pós-Graduação Faculdade Cidade Verde*, v. 4, n. 2, 2020. Disponível em: <https://revista.unifcv.edu.br/index.php/revistapos/article/view/239>.

SILVA, R. B. *Análise empírica da influência da experiência do desenvolvedor na degradação da arquitetura de software em desenvolvimento open source*. Dissertação (mathesis) — Universidade de São Paulo - Escola Politécnica, dez. 2020. Disponível em: <https://www.teses.usp.br/teses/disponiveis/3/3141/tde-20052021-130141/pt-br.php>.

SONARCLOUD. *Issues | SonarCloud Docs*. 2023. Acessado em: 30 de Junho de 2023. Disponível em: <https://docs.sonarcloud.io/digging-deeper/issues/>.

TAROUCO, H. H.; GRAEML, A. R. Governança de tecnologia da informação: um panorama da adoção de modelos de melhores práticas por empresas brasileiras usuárias. *Revista de Administração*, v. 46, p. 7–18, jan - mar 2011.

TOMOMITSU, R. H. A. *Melhoria na qualidade de projeto java com sonarqube*. Dissertação (techreport) — Universidade Estadual de Campinas, 2021. Disponível em: <https://repositorio.unicamp.br/Busca/Download?codigoArquivo=546513>.

VASCONCELOS, A. M. L. de; ROUILLER, A. C.; AND, C. Ângela F. M. *INTRODUÇÃO À ENGENHARIA DE SOFTWARE E À QUALIDADE DE SOFTWARE*. 2006. Material do CURSO DE PÓS-GRADUAÇÃO "LATO SENSU" (ESPECIALIZAÇÃO) À DISTÂNCIA MELHORIA DE PROCESSO DE SOFTWARE. Universidade Federal de Lavras - UFLA.

Appendices

.1 Troca de e-mails com o coordenador do curso de Sistemas de Informação

E-mail com assunto "Questões sobre o sistema *Sistema de gestão de curso*"

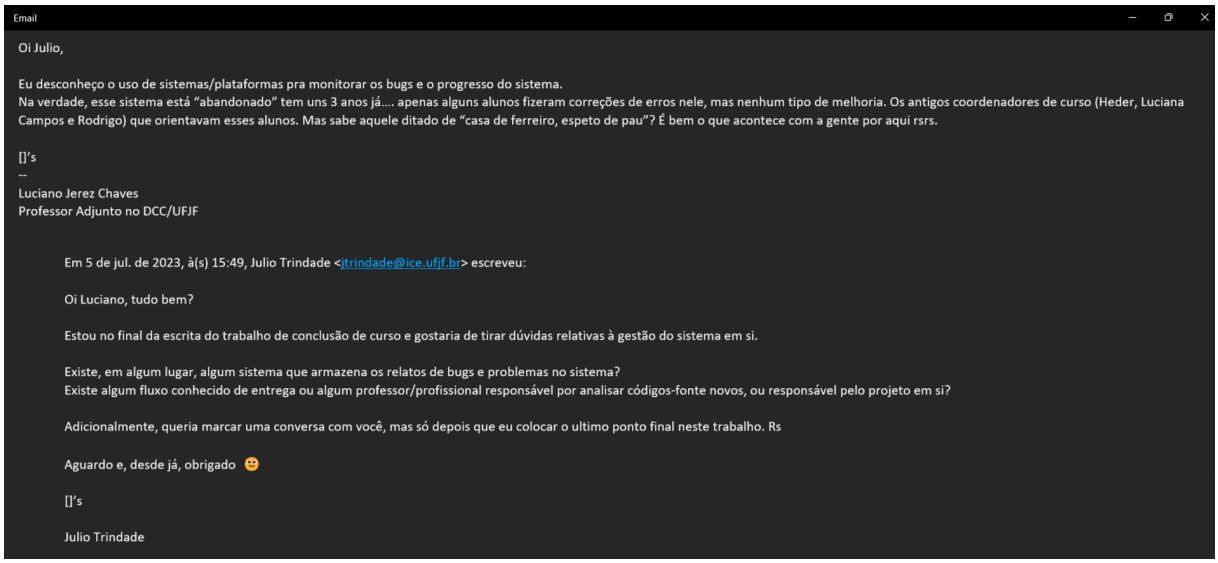

Figura 1: E-mail com coordenador de curso sobre *bugs* do sistema

E-mail com assunto "Sistema de gestão de oferta de vagas web"

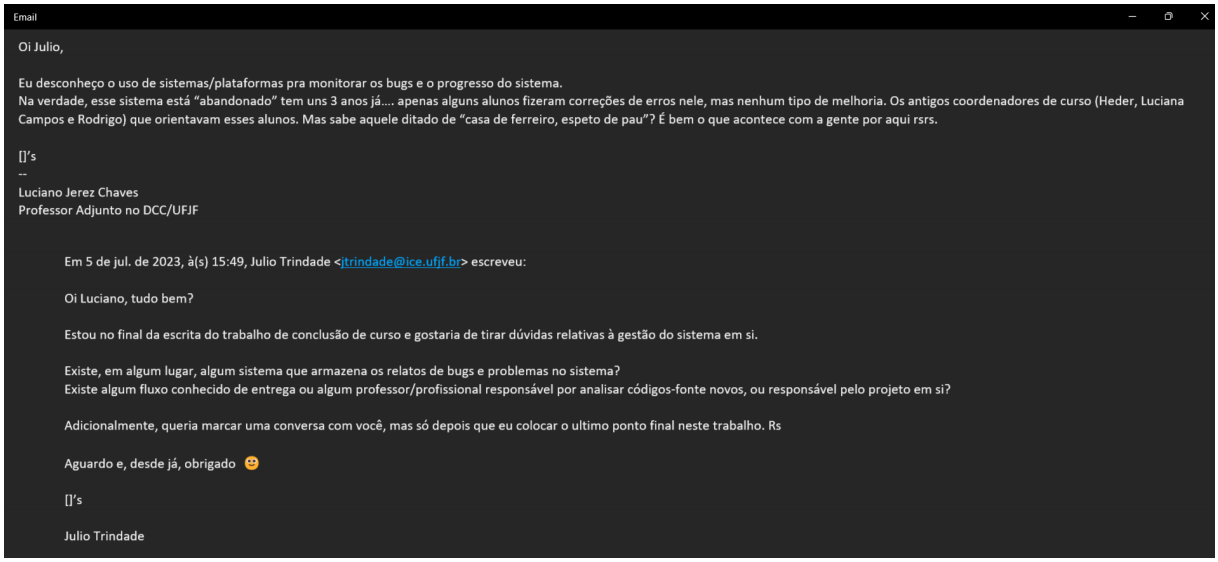

Figura 2: E-mail com coordenador de curso sobre documentação do sistema