Associações entre atributos do processo de desenvolvimento e quantidade de erros e *warnings* detectados por verificação estática

Daniel Machado Osório Pereira[1], **Lesandro Ponciano**[1]

[1]Bacharelado em Engenharia de Software
Instituto de Ciências Exatas e Informática – PUC Minas
Rua Cláudio Manoel, 1.162, Funcionários, Belo Horizonte – MG – Brasil

daniel.osorio@sga.pucminas.br, lesandrop@pucminas.br

Abstract. *The software development process consists of a series of attributes that aim to produce software with quality. A relationship is expected between the proper configuration of the development process and the results obtained in terms of quality. Software quality can be analyzed through static and dynamic verification. Static verification consists of inspections performed on the source code. For example, development environments and compilers can, when processing the code, show violations of syntax or programming logic and more serious violations that prevent the compilation of the code. This paper investigates associations between attributes of the development process and results of a static verification in terms of the classes and quantities of errors and warnings. Eighteen software development projects are analyzed in an educational interdisciplinary software work environment, which lasts one semester. Using statistical methods of correction and multiple linear regression, relationships between process attributes and the results of the static verification are studied. Among the results, we find that the size of the development cycle (sprint) affects the total amount of errors that persist in a project.*
Keywords— *Software Quality, Development Process, Static Verification*

Resumo. *Um processo de desenvolvimento de software é composto de uma série de atributos que visam produzir um software de qualidade. É esperada uma relação entre a adequada configuração do processo de desenvolvimento e os resultados obtidos em termos de qualidade de software. A qualidade de software pode ser analisada por meio de verificação estática e verificação dinâmica. A verificação estática consiste em inspeções realizadas no código-fonte. Por exemplo, ambientes de desenvolvimento e compiladores podem, ao analisar o código, mostrar violações leves de sintaxe ou lógica de programação e violações mais graves que impedem a compilação do código. Este artigo investiga associações entre atributos do processo de desenvolvimento e resultados de uma verificação estática em termos das classes e quantidades de erros e warnings. São analisados 18 projetos de desenvolvimento de software em um ambiente de trabalho interdisciplinar de software, que possuem um semestre de duração. Empregando métodos estatísticos de correlação e regressão linear múltipla, são estudadas relações entre atributos do processo e os resultados da verificação estática. Entre os resultados, pode-se apontar que o tamanho do*

ciclo de desenvolvimento (sprint) afeta a quantidade total de erro que persistem em um projeto.
Palavras chave— *Qualidade de Software, Processo de Desenvolvimento, Verificação Estática*

1. Introdução

Com os *softwares* tornando-se cada vez mais complexos e imprevisíveis, com requisitos mais sofisticados, as técnicas tradicionais de desenvolvimento têm limitações em gerar um *software* que satisfaça seus requisitos [Challagulla et al. 2008]. A habilidade de identificar a presença de problemas em um *software* é essencial para melhorar a eficácia no seu processo de desenvolvimento [Gondra 2008]. Nesse contexto, podemos observar um grande conjunto de métricas que podem ser utilizadas na relação entre atividades de *software* e problemas no *software*, úteis nas atividades de desenvolvimento de *software* [Malhotra 2014]. A qualidade de *software* pode ser analisada por meio de verificação estática e verificação dinâmica. A verificação estática consiste em inspeções realizadas no código-fonte, enquanto a verificação dinâmica opera por meio da execução do código fonte.

Linguagens de Domínio Específico (DSLs, do inglês *Domain-specific languages*) fornecem um poder de interpretação focado em um domínio de um problema específico úteis à verificação estática. As DSLs proporcionam abstrações linguísticas sobre tarefas comuns dentro de certo domínio, e com isso possibilitam que desenvolvedores foquem na lógica da aplicação, ao invés de detalhes de implementação de baixo nível. Para que desenvolvedores tirem proveito das DSLs, é necessário um Ambiente de Desenvolvimento Integrado (IDE, do inglês *Integrated Development Environment*) que, ao realizar o *parsing* dos arquivos do código, gere retornos como variáveis criadas de forma errada, referências para declaração de identificadores e até sugestões de refatoração, que podem ser classificados como erros e *warnings* [Kats and Visser 2010].

Este trabalho se insere no contexto de correlação entre atividades da Engenharia de *Software* com problemas identificados no código-fonte. **A falta de informação sobre quais atividades da produção de *software* tem maior impacto na presença de erros e *warnings* no código de projetos de *software* é o problema tratado neste estudo**. Problemas presentes em um *software* acarretam prejuízos em todo o ciclo de vida da aplicação, já que alguns deles não são detectados durante a fase de desenvolvimento [Srinivasan and Fisher 1995]. Para reduzir esses problemas, é necessário que a responsabilidade de evolução e sustentação de um sistema seja de algum *software* especializado, e não dos programadores [Zhang and Tsai 2003]. Com isso, o planejamento de projetos de *software* seria mais preciso, evitando erros na estimativa de custo e prazo, uma vez que esses não seriam afetados por defeitos inesperados [Chen et al. 2019]. Portanto, mostra-se relevante a compreensão da relação entre esses problemas com as atividades da engenharia de software.

Tendo em vista o contexto apresentado, **o objetivo deste trabalho é identificar quais atributos no processo de desenvolvimento de software impactam na presença de *errors* e *warnings* no código produzido**. Para identificar os atributos, é necessário realizar análises em projetos já executados e identificar esses atributos de cada projeto, além de analisar o código desenvolvido. Para realizar o estudo são utilizados dados do

processo de desenvolvimento e do código produzido em 18 trabalhos de desenvolvimento de *software* conduzidos em ambiente acadêmico, mas, de forma geral, com clientes reais. É realizada uma análise desses dados, identificando a correlação e modelo de regressão. Os resultados mostram uma correlação moderada entre alguns atributos do processo de desenvolvimento com o número de problemas encontrados no código, como o tamanho das *sprints* e o uso de testes automatizados. Além disso, apresenta-se um modelo de regressão múltipla que estima o número total de erros encontrados no código-fonte do projeto.

O trabalho está organizado em 7 seções. Na seção 2 está a fundamentação teórica. Nela é possível compreender todos os conceitos para a leitura da pesquisa. A seção 3 contempla os trabalhos relacionados. Já a seção 4 possui os materiais e métodos utilizados. A seção 5 apresenta os resultados encontrados no estudo. A seção 6 discute sobre as implicações do trabalho e suas limitações. Finalizando, a seção 7 percorre sobre o que foi feito no trabalho e apresenta os trabalhos futuros.

2. Fundamentação Teórica

Nesta seção são apresentados os conceitos necessários para a compreensão das atividades da Engenharia de Software. Entender o emprego desses conceitos é imprescindível para a compreensão deste estudo.

2.1. Engenharia de Software e suas atividades

O processo de *software* é um conjunto de atividades, ações e tarefas realizadas na criação do artefato. Uma atividade busca atingir um objetivo amplo, uma ação envolve um conjunto de tarefas que resultam em um artefato de *software* e uma tarefa se concentra em um objetivo específico. O processo de *software* não é rígido, possibilitando à equipe realizar o trabalho de selecionar e escolher o conjunto apropriado de ações e tarefas. O *framework* do processo de *software* engloba cinco atividades de apoio, são elas: comunicação, planejamento, modelagem, construção e entrega [Pressman and Maxim 2016]. Essas atividades são explicadas a seguir:

1. Comunicação abrange a comunicação com o cliente e todos os envolvidos no projeto. A intenção é entender os objetivos e reunir requisitos que ajudem a definir os recursos e as funções do *software*.
2. Planejamento consiste em fazer um mapa, chamado plano de projeto de *software*, que descreve as tarefas técnicas a serem conduzidas, os riscos prováveis, os recursos que serão necessários, os produtos resultantes a serem produzidos e um cronograma de trabalho.
3. Modelagem consiste em criar um esboço para que se possa ter uma ideia do todo. O engenheiro de *software* cria modelos para entender melhor as necessidades do *software* e o projeto que vai atender a essas necessidades.
4. Construção é uma atividade que combina geração de código e testes para buscar erros na codificação.
5. Entrega do *software* ao cliente, que avalia e fornece *feedback*.

2.2. Erros e *warnings* em software

Na Engenharia de Software existe uma técnica para encontrar defeitos no código-fonte sem executá-lo, que é chamada *verificação estática* [Ge et al. 2011]. A verificação

estática consiste em analisar a representação de um *software*, como seu código-fonte ou diagramas que representem o *software*, sem que o programa seja executado. Um exemplo de verificação estática é a análise estática automatizada, que consiste em uma ferramenta que percorre o código do programa e busca por potenciais problemas. Já a verificação dinâmica consiste na análise do *software* em execução, o sistema é executado com dados de teste e seu comportamento operacional é observado.

Este trabalho foca na verificação estática. Atualmente, já existem tecnologias acopladas a IDE que realizam a análise estática de código fonte e informam ao programador resultados das avaliações em alto nível. Essas tecnologias geram análises estáticas que são transformadas em definições de sintaxe declarativa, mensagens claras para que o desenvolvedor compreenda. Com isso, as IDEs conseguem diminuir o esforço de identificação dos problemas pelos desenvolvedores [Kats and Visser 2010]. Dentre essas avaliações que as IDEs retornam aos desenvolvedores, podemos destacar os erros e *warnings*.

Erros[1] são problemas na sintaxe do código que impedem a compilação do programa. Alguns exemplos de erros são: "*; expected*" para linguagens que usam ";" para indicar o fim de um comando e "*Implicitly-typed variables must be initialized*" indicando que variáveis que não possuem um tipo definido precisam ser inicializadas.

Warnings, por sua vez, são relacionados a problemas que não impedem a compilação, mas podem gerar um problema durante a execução. Alguns exemplos de *warnings* são: "*The variable 'analyze' is assigned but its value is never used*" - que indica ao desenvolvedor que o valor da variável criada não é usado; e "*Variable 'test' is used before it has been assigned a value. A null reference exception could result at runtime.*" - para variáveis que são utilizadas antes de receberem um valor, o que pode levar a uma exceção durante a execução do programa.

3. Trabalhos Relacionados

Nesta seção são apresentados os trabalhos relacionados à identificação dos atributos do processo de desenvolvimento de *software* que influenciam na presença de erros e *warnings* no código. Ainda que nem todos tratem exclusivamente deste assunto, apresentam abordagens relevantes ao presente estudo.

Moser et al. apresentam uma classificação de arquivos do projeto Eclipse, utilizando métricas relacionadas ao código para classificar esses arquivos como defeituosos ou não [Moser et al. 2008]. Após essa análise, eles utilizaram as abordagens de regressão lógica, árvores de decisão e Naive-Bayes para montar modelos de classificação e determinar qual deles teria o menor custo. Este artigo também realiza uma classificação dos problemas encontrados no código-fonte.

Gondra propõe a criação de uma rede neural artificial para classificar algoritmos como propensos a falhas ou não [Gondra 2008]. Seu estudo realizou uma análise crítica sobre as métricas usadas para classificar algoritmos e selecionou as mais importantes para treinar sua rede neural. Diferentemente desse estudo, o intuito deste trabalho é apenas identificar problemas nos algoritmos de *softwares* e relacioná-los com os atributos do

[1]Note que, na verificação dinâmica há o conceito de erro que difere do conceito de erro na verificação estática. Na verificação estática, o erro não é decorrente da execução do código com defeito na lógica de implementação.

processo de desenvolvimento.

Ge et al. propõem uma abordagem que combina verificação estática e dinâmica para encontrar problemas no código-fonte, evitando dessa forma os pontos negativos de ambas as abordagens de verificação [Ge et al. 2011]. Além de analisar o código e levantar possíveis problemas, a nova abordagem executa o programa usando dados de entrada que testam o fluxo que foi acusado como defeituoso pela verificação estática. Este artigo também utilizou verificação estática para encontrar problemas no código-fonte de sistemas.

Challagulla et al. concluíram que a garantia da qualidade do *software* de forma automatizada para sistemas dinâmicos é essencial para manter a conformidade com os requisitos levantados [Challagulla et al. 2008]. O uso da abordagem de selecionar conjuntos simples de atributos dos sistemas combinado com modelos de aprendizado de máquina apresentou um bom resultado, evidenciando que é uma boa técnica para monitorar módulos de *software*. Essa pesquisa mostra como é possível extrair informações relevantes de módulos de *software* a partir dos problemas encontrados no sistema.

Fernández Medina et al. usam as mensagens geradas pelo compilador para avaliar o aprendizado de alunos do curso de Ciência da Computação. Foi feita uma análise estatística mostrando a evolução dos alunos em cada ciclo de aprendizado, de acordo com a variação das mensagens que eram geradas pelo compilador [Fernández Medina et al. 2013]. Da mesma forma, também realizamos uma análise sobre as mensagens geradas pelo compilador.

4. Materiais e Métodos

A pesquisa realizada neste artigo tem caráter observacional e quantitativo. Possui característica observacional por se tratar de uma análise dos atributos do processo de desenvolvimento de *software* e dos problemas presentes no código-fonte, sem manipulação de variáveis, apenas observação e análise de seus valores. Essa análise consiste na busca de correlação entre esses atributos, explicitando quais atributos do processo de desenvolvimento influenciam na presença de problemas no código. A pesquisa é quantitativa pois trata da comparação entre esses atributos. O restante desta seção apresenta as etapas necessárias para a realização deste estudo, assim como os procedimentos realizados.

A Figura 1 organiza o fluxo de coleta, geração e análise dos atributos do processo de desenvolvimento e dos problemas detectados por verificação estática que foram utilizados no trabalho, bem como o método adotado para inter-relacionar atributos do processo com os resultados da verificação estática. A figura mostra que iniciamos o trabalho realizando o processamento estático do código, e com isso foram geradas classes dos erros e *warnings* encontrados. Essas classes foram integradas aos atributos de processo coletados, possibilitando realizar a correlação entre pares de variáveis e a criação de um modelo de regressão múltipla. Essas etapas são descritas ao longo desta seção.

4.1. Contexto dos Projetos de Desenvolvimento de Software

Neste trabalho, utilizou-se uma base de dados contendo informações sobre os Trabalhos Interdisciplinares de Software (TIS) realizados pelos alunos do curso de Engenharia de Software da Pontifícia Universidade Católica de Minas Gerais. Os trabalhos são realizados em diversos períodos do curso, sendo que em cada período o foco do trabalho é dife-

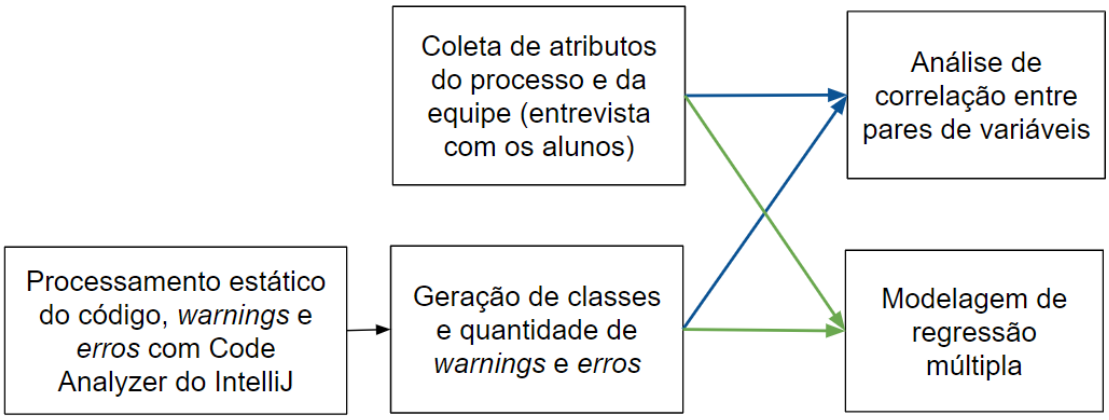

Figura 1. Fluxograma contendo as etapas da pesquisa

rente. Nesse caso, serão analisados trabalhos realizados pelos alunos do terceiro, quarto e quinto período. Essa base contém os *commits* realizados por esses alunos durante o período do trabalho, além dos contribuidores de cada trabalho e duração das *sprints*. A Tabela 1 mostra em qual período do curso o TIS foi realizado, o ano do desenvolvimento, e as linguagens que cada projeto utilizou. Ao todo, os dados utilizados envolvem 42 alunos e 6 professores, que estavam em 10 turmas de TIS ao longo de 2 anos.

O foco do TIS no terceiro período é a interação com um cliente real, logo os alunos precisam passar pelas etapas de levantamento de requisitos, construção do *Minimum Viable Product* (MVP), validação de telas e testes de usabilidade com o cliente. No quarto período a dinâmica com o cliente é semelhante, porém com um olhar para a extensão, logo os clientes são, por exemplo, igrejas, presídios e instituições de caridade. Finalmente, no quinto período não existe a necessidade de um cliente real, os alunos devem construir um aplicativo móvel que integre com *Applications Programming Interfaces* (APIs).

4.2. Dados do processo e problemas detectados por verificação estática

Os atributos do processo de desenvolvimento usado no TIS, que se relacionam com as atividades da Engenharia de Software, apresentadas na subseção 2.1, e que são considerados neste trabalho são:

- **Número de *commits* (c).** Quantidade de *commits* realizados pelos alunos durante o projeto. Relaciona-se com a atividade Construção, por ser parte do processo de geração de código.
- **Linguagens de programação utilizadas (l).** Número de linguagens utilizadas no projeto, incluindo linguagens de programação e de marcação. Relaciona-se com a atividade Planejamento, por ser necessária no momento de descrever as tarefas técnicas do projeto.
- **Uso de testes automatizados (t).** Valor booleano que indica se o projeto utilizou testes automatizados ou não. Relaciona-se com a atividade Construção, por ser uma técnica utilizada durante a geração de código para buscar erros na codificação.
- **Frequência de interação com o cliente (i).** Atributo que indica de quantas em quantas semanas a equipe se encontrava (presencialmente ou virtualmente) com o cliente para discutir sobre o andamento do projeto. Relaciona-se com a atividade

Tabela 1. Informações dos projetos

Nome dos Projetos	TIS	Ano/semestre do projeto	Linguagens de programação, marcação e estilo
Kanleitos	3	2017/2	Java, JavaScript, HTML, CSS
Projeto Incluir	3	2017/2	HTML, Java, JavaScript, CSS
Bartech	5	2018/2	TypeScript, HTML, JavaScript, CSS, Java
Calf Generator	4	2018/1	JavaScript, CSS, HTML, PHP
Will List	5	2019/1	Java, JavaScript, CSS, HTML, Kotlin
Novolharua	3	2019/1	JavaScript, CSS, HTML
Age of Philosophy	4	2017/2	C#, CSS, JavaScript, PHP, HTML
Apac	4	2017/2	C#
Pain-o-matic	4	2017/2	PHP, CSS, HTML, JavaScript
apacteca	4	2018/1	CSS, JavaScript, HTML
pectometro	4	2018/2	HTML, JavaScript, CSS
gmfonseca	4	2018/2	PHP, JavaScript, HTML, CSS
Dcbio	4	2017/2	CSS, HTML, JavaScript, PHP
Automecânica Vailante	3	2019/1	C#, PowerShell
Alpmys	5	2018/1	Java, C#, TypeScript, HTML, CSS, JavaScript
Jogo de Filosofia	3	2017/1	C#
Gestao CSF	4	2019/1	PHP, HTML
Private Class	3	2019/1	JavaScript, CSS, HTML

Comunicação, por representar a interação com o cliente, com intuito de entender os objetivos e reunir requisitos.

- **Tamanho das *sprints* (s).** Atributo que indica o número de semanas que a *sprint* durava. Relaciona-se com a atividade Planejamento, por representar a definição do tamanho do ciclo de desenvolvimento que influenciará no cronograma de trabalho.
- **Artefatos Entregues por Ciclo (a).** Valor booleano que indica se era acordado com o professor de TIS quais artefatos deviam ser entregues ao final da *sprint*. Relaciona-se com a atividade Planejamento, por representar quais artefatos seriam entregues ao final do ciclo de desenvolvimento, fazendo parte do cronograma de trabalho.
- **Ferramentas de Comunicação da Equipe (f).** Quantidade de ferramentas de comunicação que eram utilizadas pela equipe desenvolvedora. Relaciona-se com a atividade Comunicação, por representar a comunicação entre os membros do projeto.
- **Tamanho da equipe (e).** Número de desenvolvedores que participaram do projeto. Relaciona-se com a atividade Planejamento, pois o tamanho da equipe influencia no momento de descrever os riscos prováveis do projeto e o cronograma de trabalho
- **Período em que o TIS foi realizado (p).** Atributo indica em qual período (3º, 4º ou 5º) o TIS foi produzido.

A partir do estudo de dos Santos et al., que gerou uma base de dados relacional com informações dos TIS já realizados, extraímos os seguintes atributos do processo de desenvolvimento: número de *commits*, duração da *sprint*, tamanho da equipe e período em que o TIS foi realizado [dos Santos et al. 2019]. Para identificar as linguagens utilizadas foram extraídas dos repositórios GitHub dos projetos as linguagens presentes em cada projeto. Além disso, foram feitas entrevistas com os alunos que desenvolveram os projetos, para obter o restante dos atributos do processo de desenvolvimento, que não estavam presentes na base de dados relacional. As questões da entrevista foram definidas de modo a complementar os dados obtidos com informações acerca do processo de desenvolvimento utilizado. Ao todo, 16 alunos participaram das entrevistas. As perguntas propostas nas entrevistas foram as seguintes: a) No seu TIS, foram utilizados testes automatizados?; b) Com qual frequência sua equipe interagia com o cliente?; c) Quais artefatos eram entregues ao final da *sprint*?; d) Qual ferramenta de comunicação sua equipe utilizou durante a execução do projeto?

Para identificar a quantidade de erros e *warnings* foi realizada uma verificação estática da versão final do código. Dessa forma, os repositórios de cada projeto passaram pela análise da ferramenta de inspeção *Inspect Code*, que é acoplada à IDE chamada IntelliJ, que entende diversas linguagens como Java, JavaScript e TypeScript. Foi utilizado o *Inspection profile default* dessa IDE, que determina os tipo de violação no código. Os tipos de violação são: *error*, *warning* e *weak warning*. Esse trabalho considera apenas os tipos *error* e *warning*. *Weak warnings* não foram considerados, pois tratam de violações como simplificar o uso dos *imports* no código. Tais violações não geram defeitos no código, não sendo relevantes ao contexto deste trabalho. Os erros e *warnings* identificados foram agrupados em oito classes, que são:

- ***Mismatched property value.*** Erro gerado por parâmetros e propriedades com valores incompatíveis.

- ***Cannot resolve file/directory.*** Erro gerado por arquivos, classes e diretórios que não foram interpretados pela IDE.
- ***Assign variable errors.*** Erro gerado ao determinar o valor de variáveis e constantes.
- ***Redundant variables.*** *Warning* gerado por variáveis ou caracteres redundantes presentes no código.
- ***XML tag problems.*** *Warning* gerado por *tags* vazias ou *tags* que já foram descontinuadas.
- ***Missing required fields.*** Erro gerado por campos e atributos ausentes na chamada de métodos.
- ***Unused import/parameter/field.*** *Warning* gerado por importações e campos que são declarados mas não são utilizados.
- ***CSS property problems.*** *Warning* gerado por propriedades no CSS que foram descontinuadas ou que foram utilizadas de forma errada.

Considerando essas 8 classes, foram definidas as variáveis de resposta a serem estimadas pelo modelo, que se referem à ocorrência de erros e *warnings* nos projetos. As variáveis de resposta são descritas a seguir. A primeira é o **total de problemas da classe mais frequente** (P_f), que se refere ao valor de problemas na classe que exibiu a maior ocorrência de problemas. A segunda é o **total de problemas** (P_t), que é o somatório da quantidade de problemas das 8 classes, revelando a quantidade total de problemas. A terceira é o **total de problemas que são erros** (P_e), que é obtido pelo somatório das classes que descrevem erros, revelando a quantidade total de erros. Em seguida temos o **total de problemas que são *warnings*** (P_w), que é o somatório das classes que descrevem *warnings*, revelando a quantidade total de *warnings*. Por fim, a quinta e última variável a ser estimada pelo modelo é o **total de classes presentes** (P_c), que é a quantidade de classes de problemas presentes em cada projeto, visto que certos projetos não possuíam problemas de todas as classes. Os valores das classes são analisados em termos da média, desvio padrão e 95 percentil.

4.3. Método para inter-relacionar atributos do processo com resultados da verificação estática

Para entender como os atributos do processo se relacionam com os problemas encontrados no código-fonte dos projetos através da verificação estática, foi utilizado a análise de correlação e regressão usando a linguagem R-statistics. O método de correlação de Spearman é empregado para identificar quais atributos do processo de desenvolvimento se relacionam com problemas encontrados. Correlação de Spearman avalia a relação monotônica entre duas variáveis, gerando como resultado um valor que varia entre +1 e -1. Em uma relação monotônica, os valores das variáveis tendem variar juntos. Valores positivos da correlação de Spearman indicam que os valor de uma variável aumentam e os da outra variável também aumentam. Por outro lado, correlação negativa indica que, caso os valores de uma variável decresce, os valores da outra variável crescem.

Por fim, aplicou-se a técnica estatística regressão múltipla sobre a base de dados para verificar as relações entre os atributos do processo de desenvolvimento e as variáveis de resposta. A regressão múltipla é utilizada quando existem duas ou mais variáveis explicativas que influenciam o comportamento da variável resposta, sendo que essa influência é definida como uma equação. O nível de significância estatística é representado

Tabela 2. Média, Desvio Padrão e 95 Percentil das Métricas

Métricas	Média	Desvio Padrão	95-Percentil
Total de problemas da classe mais frequente (P_f)	320900.1	663683.2	1783329
Total de problemas (P_t)	393135.1	779171.5	2198196
Total de problemas que são erros (P_e)	204229.2	553851.5	940053.2
Total de problemas que são *warnings* (P_w)	188905.9	448475.6	901255
Total de classes presentes (P_e)	3.9	2.2	6.5

pelo $p - value$, que nesta pesquisa teve seu valor definido em 0.05. Outra métrica utilizada, o *Adjusted R-squared*, avalia o poder explicativo do modelo gerado levando em conta o número de variáveis explicativas presentes. Essa métrica indica a porcentagem da variação na variável resposta que as variáveis explicativas explicam coletivamente, medindo a força do relacionamento entre o modelo e a variável de resposta, em uma escala de 0 a 1. Já a estatística F foi usada para identificar a razão entre as variações do modelo, sendo que essas variações representam a distância que os dados estão dispersos da média.

5. Resultados

Os resultados obtidos no trabalho são apresentados ao longo desta seção. Primeiro, apresenta-se a análise da incidência de erros e *warnings* em cada projeto, para entender a particularidade de cada projeto. Depois é realizada a análise de correlação entre os atributos do processo com os problemas encontrados no código fonte. Por último são descritos os resultados encontrados na regressão múltipla que foi aplicada aos dados gerados.

5.1. Análise da Incidência de Erros e Warnings

Primeiramente, analisa-se a quantidade de problemas que cada projeto apresentou, como mostra a Figura 2. A diferença na quantidade de problemas encontrados é grande entre cada projeto. Isso mostra que os projetos não são uniformes entre si, havendo espaço para que a variação seja explicada por algum atributo do processo.

A Tabela 2 mostra como a média do total de classes presentes em cada projeto é baixa, 3.9, sendo que existem 8 classes. Logo, as classes de cada projeto são específicas, e dependem dos atributos do processo de desenvolvimento de *software*. Também percebemos que a média de erros é maior que a média de *warnings*, mesmo as classes sendo divididas igualmente. Isso mostra que a frequência de erros no código-fonte é maior do que a presença de *warnings*.

5.2. Análise de Correlação entre Atributos do Processo, Erros e Warnings

Analisa-se agora a correlação entre os atributos, como mostra a Tabela 3, destacando quais atributos do processo influenciaram na quantidade de problemas existentes. Nessa tabela, algumas correlações devem ser ressaltadas pelo grau de correção e pelo nível de significância.

O atributo tamanho das *sprints* (s) tem uma correlação positiva com o total de problemas da classe mais frequente (P_f), sendo $\rho(s, P_f) = 0.49$, com $p - value = 0.05$. Esse resultado mostra que *sprints* maiores estão positivamente relacionadas com o total de problemas da classe mais frequente do projeto. O atributo tamanho das *sprints* (s)

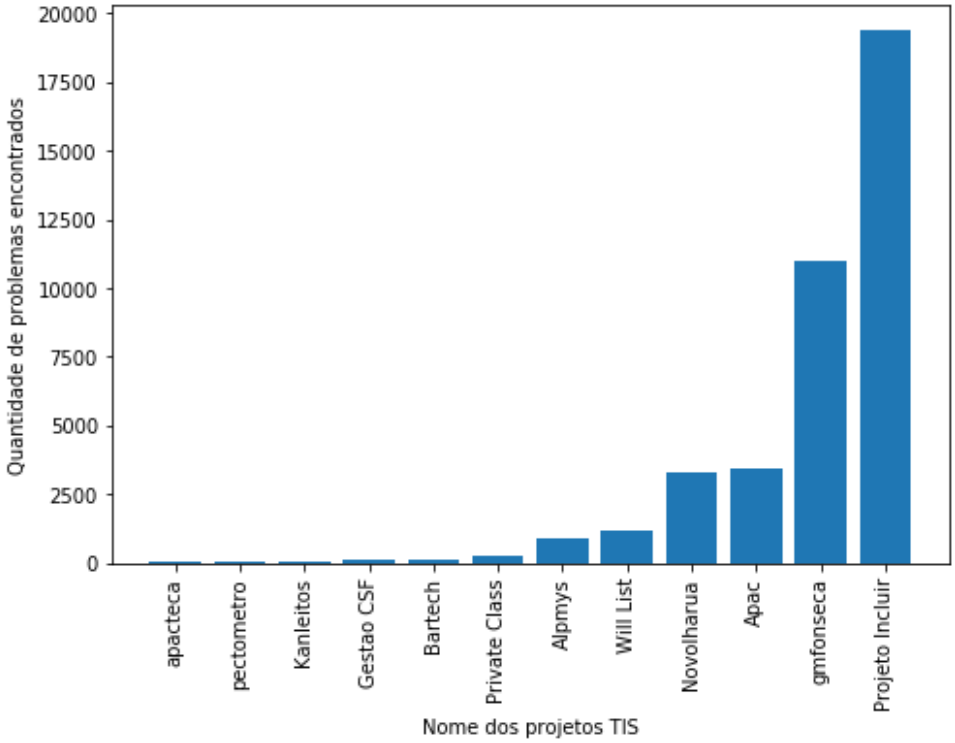

Figura 2. Quantidade de problemas encontrados para cada projeto de TIS analisado

também tem uma correlação positiva com o total de problemas que são erros (P_e, sendo $\rho(s, P_e) = 0.44$, com $p - value \leq 0.1$). Esse resultado mostra que *sprints* maiores estão positivamente relacionadas com o total de problemas que são erros. Uma das justificativas para a relação é que em *sprints* maiores os desenvolvedores têm uma quantidade maior de desenvolvimento para ser realizada, desviando a atenção da qualidade do *software* produzido. Além disso, com sprints de menor duração, as revisões realizadas ao término de cada *sprint* contribuem para um ajuste da qualidade do código, o que é feito com menos frequência quando a *sprint* é mais longa.

Já o atributo período do TIS (p) tem uma correlação negativa com a quantidade de *warnings* no projeto (P_w, sendo $\rho(p, P_w) = -0.41$, com $p - value \leq 0.1$). Esse resultado indica que o período do TIS está negativamente relacionado à quantidade de problemas que são *warnings* do projeto. É importante destacar que o período no TIS é um indicativo do quão avançado no curso o projeto está, quanto maior o período, mais próximo do término do curso, e, portanto, mais experientes os alunos que desenvolveram o projeto. Uma das justificativas para essa correlação é que a medida que os alunos vão avançando nos períodos do curso de graduação a preocupação com a qualidade do *software* também aumenta.

Para finalizar, o atributo testes automatizados (t) também apresenta uma correlação negativa com a quantidade de *warnings* no projeto (P_w, sendo $\rho(t, P_w) = -0.49$, com $p - value = 0.05$). Esse resultado indica que o uso de testes automatizados está negativamente relacionado à quantidade de problemas que são *warnings* do projeto. Uma das justificativas para essa relação é que nos projetos em que os programadores se

Tabela 3. Correlação entre os atributos de processo e as variáveis de resposta geradas

Atributos de processo	P_f	P_t	P_e	P_w	P_c
Número de *commits* (c)	-0.23	-0.23	-0.30	-0.11	-0.13
Número de linguagens (l)	0.26	0.27	0.28	0.02	-0.04
Testes automatizados (t)	-0.20	-0.24	-0.16	**-0.49***	-0.39
Interação com o cliente (i)	0.15	0.03	-0.04	0.32	0.10
Tamanho das *sprints* (s)	**0.49***	**0.41**	**0.44****	0.27	0.21
Artefato por ciclo (a)	-0.28	-0.17	-0.25	-0.17	-0.15
Ferramentas de Comunicação (f)	-0.05	-0.08	0.01	-0.29	-0.26
Tamanho da equipe(e)	0.15	0.13	0.09	0.13	0.02
Período do TIS (p)	-0.08	-0.08	0.01	**-0.41****	-0.27

Nota 1: * Spearman ρ coeficiente de correlação com $p - value = 0.05$.
Nota 2: ** Spearman ρ coeficiente de correlação com $p - value \leq 0.1$.
Nota 3: Correlações moderadas estão destacadas em negrito.

preocupam mais com a verificação dinâmica (número de testes automatizados) também há menor incidência de problemas que são *warnings*, sendo, portanto, um indicativo de preocupação com a qualidade do *software* em geral.

5.3. Modelagem usando a Regressão Múltipla

Apresenta-se agora os resultados da regressão linear múltipla. Ela foi empregada para modelar a relação entre os atributos do processo (i.e. c, l, t, i, s, a, f, e e p) e as variáveis de resposta geradas a partir dos problemas encontrados nos códigos dos projetos (i.e., P_f, P_t, P_e, P_w, P_c). Os resultados encontrados estão listados a seguir.

Na modelagem do total de problemas da classe mais frequente (P_f), obteve-se um modelo de regressão não estatisticamente significante ($F(6, 9) = 2.09, p-value = 0.19$). O valor do *Adjusted R-squared* é de 0.39. O resultado, portanto, não está relevante para um nível de significância estatística de 0.05. O que indica que não é possível gerar um modelo de atributos do processo nesse nível de significância com quantidade de dados utilizados. Na modelagem do total de problemas (P_t), também obteve-se um modelo de regressão não estatisticamente significante ($F(6, 9) = 1.19$, $p - value = 0.42$). O valor do *Adjusted R-squared* é de 0.1. O resultado, portanto, não está relevante para um nível de significância estatística de 0.05. O mesmo corre na modelagem do total de problemas que são *warnings* (P_w), obteve-se um modelo de regressão não estatisticamente significante ($F(6, 9) = 0.25, p-value = 0.96$), e total de classes presentes (P_c), em que se obteve um modelo de regressão não estatisticamente significante ($F(6, 9) = 0.40, p-value = 0.88$).

Na modelagem do total de problemas que são erros (P_e), obteve-se um modelo de regressão estatisticamente significante ($F(6, 9) = 8.25$, $p - value = 0.009$). O valor do *Adjusted R-squared* é de 0.81. O resultado, portanto, é relevante para um nível de significância estatística de 0.05. Devido a essa significância, torna-se relevante descrever a equação gerada pela regressão múltipla.

$$P_e = 8372(c) + 118219(l) + 1248755(t) - 613136(i) + 967456(s) + 748746(a) - 2643621(f) - 642370(e) - 215693(p) \quad (1)$$

A equação mostra o peso que cada atributo do processo de desenvolvimento tem sobre o número de erros encontrados no código-fonte. Logo, a equação evidencia a correlação em conjunto dos atributos do processo de desenvolvimento, diferente da correlação por pares feita anteriormente. Dessa equação, vale ressaltar os os atributos que ficaram com um peso elevado. O tamanho das *sprints*, representado como s, teve um peso positivo na equação, mostrando que *sprints* longas aumentam a quantidade de erros no código dos projetos. A quantidade de linguagens de programação utilizadas no projeto, representada como l, teve um peso positivo na equação, mostrando que quanto mais linguagens de programação forem usadas nos projetos, maior será a quantidade de erros encontrados.

6. Discussões de Implicações e Limitações

Buscando atingir o objetivo proposto, esse artigo analisou como as atividades do processo de *software* impactam no código desenvolvido e na qualidade do código. Os resultados obtidos mostram diversas relações importantes entre o processo organizado para produção de trabalhos disciplinares e a prevalência de problemas que podem ser detectados em uma verificação estática. Em particular, pode-se apontar a relação entre quantidade total de erros e o tamanho dos ciclos de desenvolvimento (*sprints*) e a relação entre a quantidade de testes automatizados e o total de *warnings* que permanecem no código. Nesse sentido, este estudo traz contribuições importantes tanto para a área de processos de *software* quanto para a área de ensino de engenharia de *software*.

Pode-se destacar que, a partir das análise dos atributos do processo de desenvolvimento conduzidas neste trabalho, é possível visualizar ações que podem ser executadas pelos professores de TIS. Por exemplo, visando aumentar a qualidade do código produzido pelos alunos, a redução no tamanho das *sprints* e o incentivo ao uso de testes automatizados podem trazer melhoria nos trabalhos desenvolvidos. O estudo conduzido também motiva que essas relações sejam investigadas no contexto de desenvolvimento de *software* comercial e *software* livre. Naturalmente, é um desafio obter acesso aos dados detalhados do processo e do *software* nesses contextos, mas a validação dessas relações nesses contextos permitiria estabelecer uma conclusão de maior generalidade. Dessa forma, os resultados obtidos motivam novas pesquisas nessa linha.

Como todo trabalho científico dessa natureza, o trabalho desenvolvido possui limitações que devem ser destacadas. A base de dados analisada possui 18 projetos, o que foi insuficiente para análises de significância estatística em alguns dos contextos de interesse. Naturalmente, produzir uma base maior e analisá-la pode aumentar a robustez das análises conduzidas. Outra limitação foi a forma de encontrar erros nos códigos analisados. O uso da IDE IntelliJ é adequado ao objetivo do trabalho, mas o estudo pode ser complementado com uma verificação dinâmica. Uma combinação da verificação estática e verificação dinâmica pode permitir uma análise mais completa da qualidade dos códigos produzidos. Além disso, o emprego de estratégias de mineração de dados, como

classificação e predição por árvore de decisão, é uma extensão natural dos modelo de regressão obtido neste estudo. Dessa forma, a partir dos resultados obtidos, novas pesquisas podem ser feitas a fim de complementar e solucionar as novas questões suscitadas no estudo.

7. Conclusões

Este trabalho colocou como objetivo entender quais atividades do processo de desenvolvimento de *software* impactam na presença de problemas no código produzido. Para isso, foram analisados projetos de *software* desenvolvidos e extraídos os dados sobre as atividades de desenvolvimento de *software* e os problemas encontrados no código fonte desses projetos, através da verificação estática realizada pela IDE. Para entender os dados obtidos foi identificada a correlação entre os atributos do processo de desenvolvimento e os problemas encontrados no código-fonte, para depois ser analisado como essa amostra de dados se comportaria em um modelo de regressão múltipla.

Essa análise mostrou que atributos como tamanho da *sprint* e testes automatizados possuem uma maior interferência na quantidade de problemas encontrados no código dos projetos. Além disso, usando regressão múltipla foi possível gerar um modelo consistente para a variável de resposta que representa o total de erros no projeto, que possibilita modelar a quantidade de erros no código-fonte dependendo dos atributos do processo de desenvolvimento utilizados. Essa análise é uma contribuição relevante para os professores que coordenam os projetos de TIS, possibilitando que eles entendam melhor quais atributos do processo de desenvolvimento devem ser alterados para alcançar uma maior qualidade dos projetos desenvolvidos.

Com a conclusão deste trabalho é possível identificar os trabalhos futuros. A análise foi feita em projetos desenvolvidos por alunos de graduação, seria interessante observar como seriam os resultados de projetos realizados no mercado, em equipes de *software*. Por ser um ambiente com características diferentes do contexto da graduação, desenvolvedores mais experientes e maior dedicação de tempo ao projeto, os resultados encontrados podem mostrar outros atributos do processo de desenvolvimento como influenciadores da qualidade do código-fonte.

Referências

Challagulla, V. U. B., Bastani, F. B., Yen, I.-L., and Paul, R. A. (2008). Empirical assessment of machine learning based software defect prediction techniques. *International Journal on Artificial Intelligence Tools*, 17(02):389–400.

Chen, X., Zhang, D., Zhao, Y., Cui, Z., and Ni, C. (2019). Software defect number prediction: Unsupervised vs supervised methods. *Information and Software Technology*, 106:161 – 181.

dos Santos, A. P., Baptista, B., Arantes, C., Ribeiro, E., Galdino, P. R., Lopes, P. P., and Barbosa, M. W. (2019). Mining undergraduate students' code repositories: insights from interdisciplinary software projects. In *Proceedings of the 10th Brazilian Workshop on Agile Methods*, ICML '07. Springer.

Fernández Medina, C., Pérez Pérez, J. R., Alvarez, V., and Ruíz, M. (2013). Assistance in computer programming learning using educational data mining and learning analytics. In *Proceedings of the 10th Brazilian Workshop on Agile Methods*, pages 237–242.

Ge, X., Taneja, K., Xie, T., and Tillmann, N. (2011). Dyta: Dynamic symbolic execution guided with static verification results. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 992–994, New York, NY, USA. Association for Computing Machinery.

Gondra, I. (2008). Applying machine learning to software fault-proneness prediction. *Journal of Systems and Software*, 81(2):186–195.

Kats, L. C. and Visser, E. (2010). The spoofax language workbench: Rules for declarative specification of languages and ides. *SIGPLAN Not.*, 45(10):444–463.

Malhotra, R. (2014). Comparative analysis of statistical and machine learning methods for predicting faulty modules. *Applied Soft Computing*, 21:286 – 297.

Moser, R., Pedrycz, W., and Succi, G. (2008). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 181–190, New York, NY, USA. ACM.

Pressman, R. and Maxim, B. (2016). *Engenharia de Software-8ª Edição*. McGraw Hill Brasil.

Srinivasan, K. and Fisher, D. (1995). Machine learning approaches to estimating software development effort. *IEEE Transactions on Software Engineering*, 21(2):126–137.

Zhang, D. and Tsai, J. J. (2003). Machine learning and software engineering. *Software Quality Journal*, 11(2):87–119.