UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

José Lucas da Costa Silva

Minimizando a complexidade do código e aprimorando a capacidade de manutenção por meio do gerenciamento de variabilidade com abordagem agnóstica à linguagem

Recife

2025

José Lucas da Costa Silva

Minimizando a complexidade do código e aprimorando a capacidade de manutenção por meio do gerenciamento de variabilidade com abordagem agnóstica à linguagem

Trabalho apresentado ao Programa de Graduação em Engenharia da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Engenharia da Computação.

Área de Concentração: Engenharia de software

Orientador (a): Breno Alexandro Ferreira de Miranda

Recife

2025

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Silva, José Lucas da Costa.
Minimizando a complexidade do código e aprimorando a capacidade de manutenção por meio do gerenciamento de variabilidade com abordagem agnóstica à linguagem / José Lucas da Costa Silva. - Recife, 2025.
47 p. : il., tab.

Orientador(a): Breno Alexandro Ferreira de Miranda
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Engenharia da Computação - Bacharelado, 2025.
Inclui referências.

1. Variabilidade de software. 2. Teste de variabilidade. 3. Feature flags. 4. Pré-processadores. 5. Complexidade de código. I. Miranda, Breno Alexandro Ferreira de. (Orientação). II. Título.

000 CDD (22.ed.)

JOSÉ LUCAS DA COSTA SILVA

Minimizando a complexidade do código e aprimorando a capacidade de manutenção por meio do gerenciamento de variabilidade com abordagem agnóstica à linguagem

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de bacharel em Engenharia da Computação.

Aprovado em: 04/04/2025

BANCA EXAMINADORA

_____

Prof. Dr. Breno Alexandro Ferreira de Miranda (Orientador)

Universidade Federal de Pernambuco

_____

Prof. Dr. Paulo Henrique Monteiro Borba (Examinador Interno)

Universidade Federal de Pernambuco

Eu dedico essa dissertação para toda minha família e amigos.

AGRADECIMENTOS

Primeiramente, agradeço minha família, sou profundamente grato pelo amor, incentivo e apoio constante.

Agradeço aos meus professores pelas conversas inspiradoras sobre minha pesquisa, que foram fundamentais para o desenvolvimento deste trabalho. Aos meus amigos, deixo minha eterna gratidão por me encorajarem e por estarem ao meu lado durante esta jornada. A minha amada pelo apoio e criticas construtivas durante esse processo.

Agradeço à UFPE pelo ambiente e recursos que permitiram o crescimento dos meus conhecimentos. Agradeço também aos membros da banca pela disponibilidade e contribuição na avaliação desta proposta.

RESUMO

Gerenciar a variabilidade de software é essencial para linhas de produtos de software, e práticas como feature toggles e pré-processadores são amplamente adotadas para lidar com essa complexidade. No entanto, essas abordagens frequentemente aumentam a complexidade do código, elevando os custos de manutenção e a dívida técnica. Neste artigo, apresentamos o FLAG, uma abordagem independente de linguagem, desenvolvida para facilitar o gerenciamento da variabilidade e reduzir a complexidade do código. Avaliamos o impacto do FLAG em 17 projetos de código aberto que abrangem cinco linguagens diferentes, comparando métricas de complexidade antes e depois de sua aplicação. Os resultados demonstram que o FLAG reduz significativamente a complexidade do código em todos os projetos e linguagens analisados, evidenciando seu potencial para melhorar a manutenção e a evolução de software.

Palavras-chaves: Variabilidade de software, teste de variabilidade, feature flags, pré-processadores, complexidade de código.

ABSTRACT

Managing software variability is essential for software product lines, and practices such as feature toggles and preprocessors are widely adopted to deal with this complexity. However, these approaches often increase code complexity, raising maintenance costs and technical debt. In this paper, we introduce FLAG, a language-agnostic approach designed to facilitate variability management and reduce code complexity. We evaluated the impact of FLAG on 17 open-source projects covering five different languages, comparing complexity metrics before and after its application. The results show that FLAG significantly reduces code complexity in all the projects and languages analyzed, highlighting its potential to improve software maintenance and evolution.

Keywords: Software variability, variability testing, feature flags, preprocessor directives, code complexity.

LISTA DE FIGURAS

LISTA DE CÓDIGOS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ASTs	*Abstract syntax trees*
CiC	*Complexidade Ciclomatica*
CLI	*Command Line Interface*
CoG	*Complexidade Cognitiva*
IDE	*Integrated Development Environment*
IQR	*Interquartile Range*
SPLE	*Software Product Line Engineering*
SPLs	*Software Product Lines*

SUMÁRIO

1 INTRODUÇÃO

Neste capítulo são apresentados o contexto e a motivação deste trabalho de conclusão de curso. Posteriormente, o problema da pesquisa e os objetivos são evidenciados. Por último, são apresentadas a metodologia e a organização do documento.

1.1 CONTEXTO E MOTIVAÇÃO

Linhas de Produtos de Software (*Software Product Lines* (SPLs)) e a Engenharia de Linhas de Produtos de Software (*Software Product Line Engineering* (SPLE)) são estratégias fundamentais para elevar a qualidade do software e reduzir os custos de manutenção. A promessa de ganhos econômicos e aumento da competitividade no mercado tem despertado significativo interesse entre pesquisadores acadêmicos e profissionais da indústria de software (METZGER; POHL, 2014). A SPLE oferece um arcabouço disciplinado para a criação e sustentação de SPLs, aproveitando recursos de desenvolvimento compartilhados (POHL; BÖCKLE; LINDEN, 2005). Para organizações[1] que desenvolvem uma variedade de produtos de software relacionados, adotar SPLs proporciona benefícios substanciais: aumento da produtividade, maior qualidade dos produtos, redução do tempo de lançamento no mercado e diminuição dos custos de desenvolvimento (CLEMENTS; NORTHROP, 2002; LINDEN; SCHMID; ROMMES, 2007).

O gerenciamento da variabilidade de software é essencial para qualquer SPL, sendo um fator determinante para o desenvolvimento eficiente de famílias de produtos relacionadas. Essa variabilidade pode ser abordada em diversas fases: levantamento de requisitos, planejamento arquitetural, projeto detalhado e implementação, cada uma oferecendo mecanismos distintos para incorporar a variabilidade de maneira coesa à SPL (CHEN; BABAR, 2011; BACHMANN; CLEMENTS, 2005). Idealmente, o software deveria evoluir por todas essas etapas mantendo a variabilidade; no entanto, adaptar produtos existentes para uma SPL é frequentemente mais econômico do que construir uma do zero (BASTOS et al., 2017; BREIVOLD; LARSSON; LAND, 2008; DANIEL; OMOBOLANLE, 2020). Dois fatores alimentam essa preferência: primeiro, é difícil prever antecipadamente a necessidade de uma SPL—produtos relacionados frequentemente surgem de forma orgânica a partir de uma oferta inicial. Segundo, iniciar do zero implica descartar o conhecimento consolidado nas bases de código existentes (BREIVOLD; LARSSON; LAND, 2008; NORTHROP et al., 2007).

[1] http://splc.net/fame.html

Feature flags, opções de configuração e diretivas de pré-processador são amplamente adotadas para atingir esse objetivo (RAHMAN, 2023; PR Newswire, 2020). Essas abordagens permitem que os desenvolvedores ajustem dinamicamente a funcionalidade—como alternar recursos. No entanto, essa flexibilidade implica um custo: à medida que as configurações se multiplicam, a complexidade do código aumenta, os testes tornam-se logisticamente desafiadores e o esforço de manutenção cresce. O crescimento exponencial das combinações possíveis—por vezes alcançando milhares (COHEN; DWYER; SHI, 2006), como observado em projetos como o kernel do Linux (TARTLER et al., 2014)—aumenta a pressão sobre as equipes de desenvolvimento.

Pesquisadores têm investigado extensivamente a variabilidade em SPLs nos níveis de modelagem e arquitetura. Alguns (SCHMID; JOHN, 2004; LIEBIG et al., 2010) utilizam pré-processadores como mecanismos de variabilidade baseados em ramificações no código, enquanto estudos mais recentes (JÉZÉQUEL; KIENZLE; ACHER, 2022) fazem uso de feature toggles ou, de forma ainda mais flexível, de ambas as técnicas. Como uma abordagem emergente para o gerenciamento da variabilidade, o FLAG propõe-se a ser uma alternativa independente de linguagem frente às abordagens tradicionais que tendem a aumentar a complexidade do código.

Além dos esforços nos níveis de modelagem e arquitetura, nas últimas décadas foram estabelecidas técnicas de desenvolvimento de software (MAHMOOD; OXLEY, 2011; AMIN; OXLEY, 2010) que visam reduzir a complexidade percebida e facilitar a escalabilidade dos sistemas. Esse conjunto de abordagens busca utilizar a programação orientada a objetos e suas derivações para produzir sistemas com baixo acoplamento e alta coesão (SHARVIT, 2022). Tal solução tem sido amplamente empregada em sistemas modernos, alcançando resultados significativos. No entanto, não se trata de uma solução universal capaz de resolver, simultaneamente, os problemas de escalabilidade e de variabilidade (BLACK, 2013; SINGH; CHOUHAN; VERMA, 2021).

Neste artigo, apresentamos o FLAG, uma abordagem projetada para lidar com a variabilidade de software por meio de diretivas de pré-processamento. Diferentemente das abordagens convencionais, o FLAG utiliza diretivas de pré-processador para reestruturar blocos de código durante a compilação, promovendo maior legibilidade e controle da complexidade. Além disso, o FLAG estabelece um processo estruturado, permitindo que desenvolvedores adicionem ou removam blocos de funcionalidades de maneira programática. Essa capacidade pode simplificar a manutenção e reduzir a dívida técnica, assegurando que a base de código se adapte eficientemente conforme as demandas do projeto evoluem.

Para investigar a eficácia da nossa abordagem, conduzimos um experimento envolvendo cinco linguagens de programação em 17 projetos de código aberto. Nossa avaliação consistiu na coleta de duas métricas de complexidade do código antes e depois da aplicação do FLAG como mecanismo de gerenciamento de variabilidade. Em seguida, realizamos uma análise comparativa para avaliar o impacto do FLAG na redução da complexidade do código, proporcionando uma compreensão mais aprofundada de seus benefícios na manutenção e evolução de software.

1.2 ORGANIZAÇÃO DO DOCUMENTO

Este projeto foi estruturado em capítulos conforme a seguinte organização:

- **Capítulo 2** - Traz os estudos relacionados a gerenciamento de variabilidade e complexidade de software;
- **Capítulo 3** - Retrata em detalhes o funcionamento interno da abordagem proposta;
- **Capítulo 4** - Aborda em detalhes como o experimento para a avaliação da abordagem proposta foi realizada;
- **Capítulo 5** - Aborda os resultados das medições e capturas de métrica;
- **Capítulo 6** - Sumariza a pesquisa e os resultados e manifesta os trabalhos futuros

2 TRABALHOS RELACIONADOS

Nesta seção, apresentamos uma visão geral do estado da arte em estudos sobre complexidade de software e manutenção. Também posicionamos nosso trabalho no contexto das pesquisas atuais em gestão de variabilidade de software, ou simplesmente configuração de software (MAHDAVI-HEZAVEH; FATIMA; WILLIAMS, 2024).

2.1 COMPLEXIDADE DE SOFTWARE E MANUTENÇÃO

Em um estudo envolvendo 450 projetos de código aberto hospedados na plataforma SourceForge[1], Midha (MIDHA, 2008) investigou o impacto das mudanças na complexidade estrutural—medida por meio da Complexidade Ciclomática de McCabe (MCCABE, 1976) e do Esforço de Halstead—na manutenção de software (HALSTEAD, 1977). Os resultados indicaram que o aumento da complexidade estrutural está positivamente associado ao número de defeitos e ao tempo necessário para corrigi-los, sugerindo que bases de código mais complexas demandam maior esforço de manutenção. Além disso, constatou-se que a complexidade dificulta a contribuição de novos desenvolvedores, o que agrava ainda mais o desafio da manutenção, uma vez que menos pessoas estão dispostas a interagir com um código intricado. Tais evidências reforçam que códigos complexos aumentam a carga cognitiva exigida para sua compreensão e modificação, sendo que desenvolvedores gastam mais de 50% do tempo de manutenção apenas para entender o código.

Antinyan et al. (ANTINYAN; STARON; SANDBERG, 2017) realizaram uma pesquisa com 100 engenheiros de software da indústria escandinava, com o objetivo de explorar os efeitos da complexidade do código no tempo de manutenção e na qualidade interna do software. Os achados indicaram que fatores como arquitetura deficiente, estruturas de controle profundas e encadeamento excessivo são os principais gatilhos de complexidade no desenvolvimento de software, exigindo esforço significativamente maior para manutenção. O estudo também destacou que a complexidade afeta negativamente atributos críticos de qualidade como legibilidade, compreensibilidade e modificabilidade, os quais são essenciais para uma manutenção eficiente, pois reduzem o tempo e o esforço cognitivo necessário para essa tarefa.

Ogheneovo (OGHENEOVO et al., 2014) demonstrou uma correlação direta entre o aumento das linhas de código—um dos principais indicadores de complexidade—e a elevação dos custos

[1] https://sourceforge.net/

de manutenção. Por exemplo, o sistema operacional Windows teve sua base de código expandida de 4–5 milhões de linhas na versão NT 3.0 (lançada em 1993) para 80 milhões de linhas no Windows 8 (lançado em 2012), com os custos de manutenção estimados aumentando de 28 bilhões para 192 bilhões de dólares norte-americanos no mesmo período. Benaroch et al. (BENAROCH; LYYTINEN, 2022), por sua vez, realizaram uma análise econométrica com três anos de dados de manutenção de 426 sistemas críticos de empresas listadas na Fortune 100. Além do aumento nos custos de manutenção, fatores qualitativos como produtividade, complexidade e instabilidade das equipes de desenvolvimento também foram identificados como agravantes.

Diversos estudos (MAHMOOD; OXLEY, 2011) exploraram técnicas inovadoras de implementação de software, aproveitando a programação orientada a objetos e suas variações para reduzir a complexidade e aumentar a coesão do código (SINGH; CHOUHAN; VERMA, 2021). Essa abordagem atenua a complexidade tipicamente concentrada em um único componente ao distribuí-la entre múltiplos componentes, o que contribui para a escalabilidade de Linhas de Produto de Software. No entanto, a gestão da variabilidade continua sendo um desafio, pois cada componente implementado implica em um ponto de decisão, seja em tempo de compilação ou de execução. Em contraste, FLAG pode ser integrada a essas técnicas, oferecendo uma solução ainda mais robusta.

2.2 GERENCIAMENTO DE VARIABILIDADE

Meinicke et al. (MEINICKE et al., 2020) destacam a distinção entre opções de configuração e feature toggles, embora ambas utilizem mecanismos semelhantes. As opções de configuração podem utilizar estruturas condicionais `if/else` ou diretivas de pré-processadores, sendo o fator comum o uso de arquivos de configuração para armazenar e recuperar o estado das funcionalidades. FLAG opera de forma semelhante ao utilizar um arquivo para registrar o estado das funcionalidades. Contudo, diferentemente das abordagens tradicionais baseadas em condicionais ou pré-processadores, FLAG utiliza anotações diretamente no código para tratar a variabilidade. Como demonstrado na Seção 5, essa abordagem reduz a complexidade geral e o esforço manual necessário para gerenciar funcionalidades ao longo da base de código.

O uso de pré-processadores (LIEBIG et al., 2010; MEDEIROS et al., 2015), tanto em linhas de produto quanto na indústria em geral, mostrou-se eficaz na gestão de variabilidade. Entretanto, sistemas com alta variabilidade enfrentam sérios problemas de escalabilidade (MEDEIROS et al.,

2017), o que afeta diretamente a ocorrência de erros e eleva o esforço de manutenção. Embora funcione de maneira semelhante, FLAG se destaca por oferecer não apenas uma maneira sistemática de adicionar ou remover blocos de variabilidade de forma programática, mas também por adotar uma abordagem independente de linguagem.

Feature toggles (RAHMAN et al., 2016) são atualmente o método mais comum de se introduzir variabilidade em diferentes linguagens de programação. Esses mecanismos evoluíram para permitir operação local ou remota, com técnicas como testes A/B sendo amplamente utilizadas nas últimas décadas. Contudo, assim como os pré-processadores, os feature toggles carecem de facilidade de manutenção, e há estudos que os associam a defeitos e à dívida técnica. Em contrapartida, FLAG oferece um mecanismo nativo para gerir pontos de variabilidade por meio da transformação programática do código (FEITELSON; FRACHTENBERG; BECK, 2013).

Pesquisadores da Uber demonstraram que, embora feature flags facilitem o desenvolvimento ágil e a entrega contínua, elas também podem gerar elevada dívida técnica quando permanecem ativas após seu uso, resultando em código morto e desafios de manutenção—um problema que técnicas tradicionais de eliminação de código morto e refatorações anteriores, especialmente aquelas direcionadas a diretivas de pré-processadores, enfrentaram dificuldades para resolver. Abordagens como a técnica de refatoração offline apresentada pelo PIRANHA (RAMANATHAN et al., 2020) demonstraram potencial para automatizar a limpeza de código em sistemas de larga escala. No entanto, essas soluções geralmente dependem de intervenções periódicas, e não de uma gestão contínua. Em contraste, FLAG oferece uma abordagem online para controle da dívida técnica associada a flags, promovendo assim uma redução contínua dessa dívida ao longo do tempo.

Mecanismos de variabilidade têm se desenvolvido com abordagens distintas, adequadas a contextos específicos de desenvolvimento. Por exemplo, o widget desenvolvido por Lopes et al. (LOPES; AMORIM; FERREIRA, 2021) representa uma contribuição significativa para a gestão de variabilidade em ambientes low-code, facilitando a adição intuitiva e a remoção prática de mecanismos de variabilidade. Embora essa abordagem seja adequada ao seu escopo específico, FLAG busca atender a um conjunto mais amplo de cenários, suportando múltiplas linguagens de programação e tipos de arquivos. Essa versatilidade posiciona FLAG como uma abordagem complementar, oferecendo maior flexibilidade e adaptabilidade para diferentes práticas de desenvolvimento, inclusive fora do ecossistema low-code, embora em um nível relacionado, mas substancialmente distinto.

Souza et al. (SOUZA et al., 2024) demonstraram que a Transplantação de Software (Software Transplantation - ST), utilizada em Linhas de Produto de Software, pode melhorar significativamente a adoção e a manutenção dessas linhas ao integrar funcionalidades entre diferentes bases de código. Inspirada por essa linha de pesquisa, FLAG compartilha características semelhantes, embora o foco do nosso trabalho esteja na gestão de variabilidade, enquanto o FOUNDRY concentra-se na migração de funcionalidades.

Abaixo, na Tabela 1, apresentamos uma visão consolidada comparando FLAG com outras soluções em termos de independência de linguagem, aumento de complexidade e dívida técnica.

Tabela 1 – Comparação de abordagens FLAG vs. Outros.

Técnica	**Language Agnosticism***	**Complexidade**	**Débito Técnico**
Configuration Options	✔	MÉDIA	ALTO
Pré-processors	✖	MÉDIA	ALTO
Feature toggles	✔	MÉDIA	ALTO
Piranha	✖	-	-
Outsystems Plugin	✖	BAIXO	MÉDIO
Foundry	✔	-	-
FLAG	✔	BAIXO	BAIXO

Language Agnosticism*: Agnóstico em relação à linguagem.

Fonte: Elaborada pelo autor (2025)

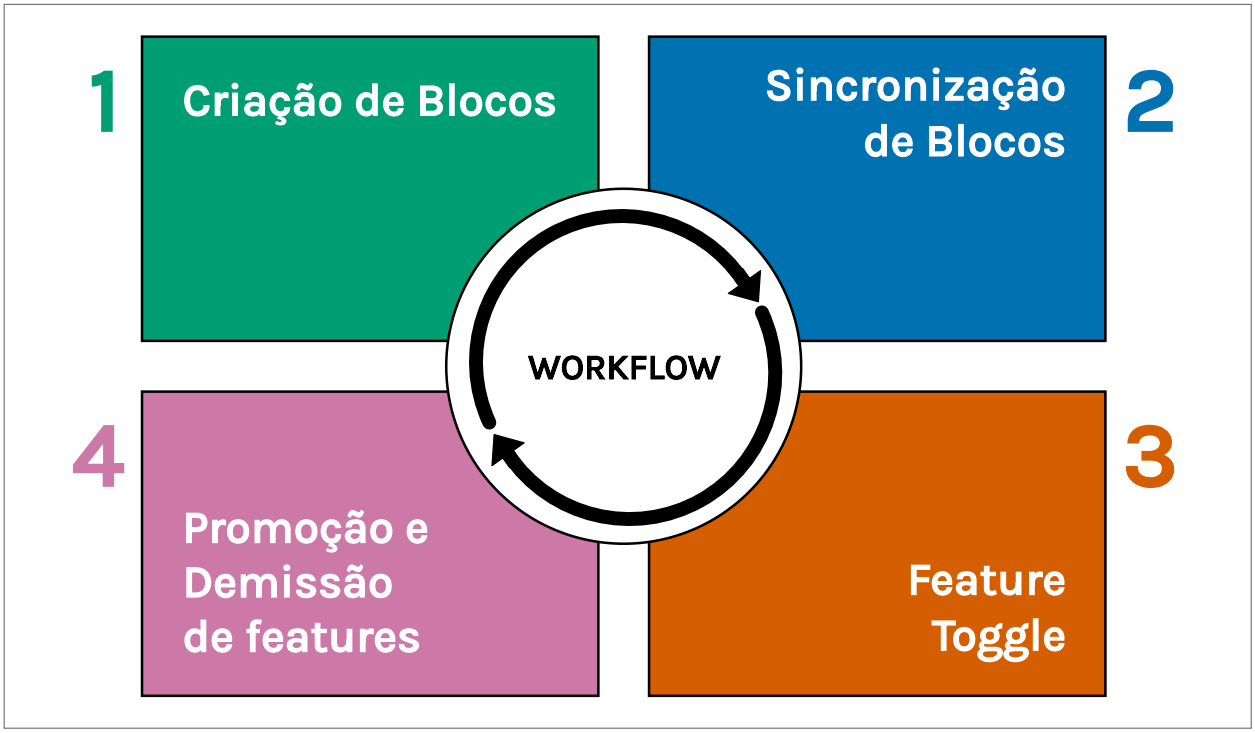
3 PROPOSTA

Nesta seção, apresentamos o funcionamento detalhado do FLAG, uma abordagem de que visa facilitar a gestão de variabilidade em sistemas de software. A seguir, descrevemos os principais mecanismos oferecidos por FLAG, incluindo a criação e sincronização de blocos de variabilidade, alternância de funcionalidades (*feature toggling*), bem como os processos de promoção e demoção de funcionalidades. Além disso, discutimos o suporte à gestão de arquivos que não permitem anotações por comentários, destacando a flexibilidade e adaptabilidade do FLAG em diferentes contextos de desenvolvimento.

FLAG fornece uma interface de linha de comando (*Command Line Interface* (CLI)) implementada em Go (Golang), que aproveita a infraestrutura do Git[1] para simplificar o gerenciamento de variabilidade de software. Em vez de substituir o Git, FLAG potencializa sua usabilidade ao oferecer um gerenciamento eficiente de feature toggles representados por diretivas de pré-processador. FLAG facilita o acionamento e desativação de funcionalidades e configurações, permitindo aos desenvolvedores controlar a variabilidade sem modificar diretamente o código-fonte. A Figura 1 apresenta uma visão geral do fluxo de trabalho do FLAG, ilustrando o ciclo de vida de uma funcionalidade utilizando seu mecanismo de variabilidade. Esse ciclo envolve a criação da variabilidade por meio de diretivas específicas de pré-processador, a fase de uso em que as funcionalidades são sincronizadas e alternadas, e a remoção do mecanismo de variabilidade quando este deixa de ser necessário.

[1] https://git-scm.com/

Figura 1 – Fluxo de trabalho do FLAG



Fonte: Elaborada pelo autor (2025)

3.1 CRIAÇÃO DE BLOCOS

A abordagem do FLAG para o gerenciamento da variabilidade de funcionalidades substitui estruturas condicionais tradicionais como `if/else` por delimitadores personalizáveis baseados em diretivas de pré-processador. A diretiva *@block* define a estrutura do bloco, enquanto as diretivas `@if`, `@else if` e `@else` representam instruções condicionais. As palavras reservadas `"and"` e `"or"` permitem maior flexibilidade na criação dos blocos. A diretiva *@end* marca o término do segmento de código. Como ilustrado no Listing 1, todas as condicionais podem coexistir durante o desenvolvimento. Os blocos de variabilidade podem operar em modo de desenvolvimento, exibindo todas as condições e seus conteúdos, ou em modo ativo, no qual o FLAG avalia as condições presentes e exibe apenas o conteúdo correspondente. Essa abordagem permite que os desenvolvedores mantenham trechos de código habilitados e desabilitados por funcionalidades, com suporte a funcionalidades aninhadas e maior flexibilidade para futuras alterações.

Código Fonte 1 – Diretivas do FLAG.

```
renderCheckoutButton() {
  // @block //
  // @if hide-checkout-button //
  return null
  // @else if big-checkout-button //
  return renderReallyBigCheckoutButton()
  // @else //
  return renderDefaultCheckoutButton()
  // @end //
}
```

Fonte: Elaborado pelo autor (2025)

O FLAG exige essas anotações para garantir a identificação precisa do código, algo que, apesar das tentativas de automatização (RUBIN; CHECHIK, 2013), ainda carece de precisão e demanda grande esforço de adaptação. Dessa forma, o estado da arte permanece predominantemente manual (KRÜGER; BERGER; LEICH, 2019), exigindo conhecimento substancial do domínio do problema. Pode-se argumentar que criar um novo bloco envolve esforço comparável à criação de estruturas semelhantes, como diretivas de pré-processador ou feature toggles. No entanto, após a criação do bloco, o FLAG automatiza o processo, reduzindo o atrito e a complexidade no gerenciamento de pontos de variabilidade.

Antes de iniciar a criação de blocos, é necessário inicializar o FLAG com o comando `init`. Este comando é responsável por criar todos os arquivos necessários para o funcionamento, incluindo o armazenamento do estado das funcionalidades presentes na base de código, os marcadores de comentários suportados e uma pasta com as informações dos blocos.

3.2 SINCRONIZAÇÃO DE BLOCOS

O FLAG inclui um mecanismo de sincronização para gerenciar o estado e o conteúdo das funcionalidades em arquivos. Esse processo é iniciado com o comando sync, que utiliza o `git` para identificar arquivos modificados, não rastreados ou excluídos. Em seguida, o FLAG cria, atualiza ou remove as informações de rastreamento interno, assegurando a consistência entre o estado atual do arquivo e os metadados. Quando o FLAG detecta um bloco pela primeira vez, atribui a ele um identificador único. Caso contrário, remove os metadados associados quando o bloco não é mais encontrado. O Listing 2 exibe um bloco de código sincronizado.

Código Fonte 2 – Um bloco sincronizado.

```
renderCheckoutButton() {
  // @block 92bca598fdacfe887ed78bcea //
  // @if hide-checkout-button //
  return null
  // @else if big-checkout-button //
  return renderReallyBigCheckoutButton()
  // @else //
  return renderDefaultCheckoutButton()
  // @end //
}
```

Fonte: Elaborado pelo autor (2025)

3.3 TOGGLE DE FEATURES

O mecanismo de alternância de funcionalidades do FLAG transforma blocos de código com base no estado de um conjunto de funcionalidades. Com o comando `toggle`, uma lista de funcionalidades é exibida ao usuário, que pode ativá-las (`ON`) ou desativá-las (`OFF`). Como discutido na subseção de Criação de Blocos, o conteúdo dos blocos pode variar conforme o modo de operação. O Listing 3 apresenta o conteúdo de um bloco após a avaliação da funcionalidade. Caso nenhuma condição seja satisfeita, o bloco é esvaziado. Essa abordagem garante que a base de código reflita sempre a configuração ativa. Ao adaptar dinamicamente o código visível, o FLAG contribui para o controle da complexidade e ajuda a mitigar a dívida técnica ao longo do tempo.

Código Fonte 3 – Bloco após "hide-checkout-button"ser ligada.

```
renderCheckoutButton() {
  // @block 92bca598fdacfe887ed78bcea //
  return null
  // @end //
}
```

Fonte: Elaborado pelo autor (2025)

3.4 PROMOÇÃO/DEMISSÃO DE FUNCIONALIDADES

O FLAG fornece um mecanismo sistemático para o gerenciamento de blocos de variabilidade durante o desenvolvimento e o ciclo de vida de funcionalidades. Com os comandos `promote`

ou demote, o usuário pode integrar ou remover permanentemente uma funcionalidade da base de código. As subseções a seguir exploram ambos os mecanismos, utilizando como exemplo o Listing 4.

Código Fonte 4 – Bloco antes da funcionalidade "hide-checkout-button"ser promovida.

```
renderCheckoutButton() {
  // @block 92bca598fdacfe887ed78bcea //
  // @if hide-checkout-button //
  return null
  // @else //
  return renderDefaultCheckoutButton()
  // @end //
}
```

Fonte: Elaborado pelo autor (2025)

3.4.1 Promoção

Durante a fase de desenvolvimento de uma linha de produto, as funcionalidades são avaliadas constantemente quanto à reutilização e integração com outras. Com isso em mente, o FLAG oferece um mecanismo de promoção que incorpora uma funcionalidade à base de código. Nessa operação, o FLAG percorre todos os blocos de código, removendo a referência à funcionalidade e incorporando seu conteúdo ao código-fonte. Esta operação pode ser dividida em dois cenários: (1) a funcionalidade promovida é a única presente na estrutura condicional. Nesse caso, a estrutura condicional é removida e seu conteúdo é incorporado diretamente ao código; (2) a estrutura condicional possui outras funcionalidades. Nesse caso, apenas a menção à funcionalidade promovida é removida, assumindo-se que ela sempre será avaliada como TRUE.

Código Fonte 5 – Bloco depois da funcionalidade "hide-checkout-button"ser promovida.

```
renderCheckoutButton() {
    return null
}
```

Fonte: Elaborado pelo autor (2025)

3.4.2 Demissão

De forma semelhante à promoção, embora com menor frequência, pode ser necessário remover uma funcionalidade da base de código ao longo da evolução da linha de produto. Internamente, o FLAG opera de forma análoga à promoção, porém, o conteúdo da funcionalidade é descartado em vez de incorporado.

Como demonstrado nos Listings 5 e 6, o FLAG é capaz de limpar o bloco de código — ou seja, o mecanismo de variabilidade — quando este se torna desnecessário, atualizando o conteúdo do bloco e as informações de rastreamento, contribuindo para a manutenção da base de código. Uma abordagem offline, o Piranha (RAMANATHAN et al., 2020), demonstrou reduções significativas em funcionalidades não utilizadas. Em contrapartida, FLAG, como abordagem online, atua de forma preventiva, evitando o acúmulo de dívida técnica e garantindo a manutenibilidade do sistema.

Código Fonte 6 – Bloco depois da funcionalidade "hide-checkout-button"ser demitida.

```
renderCheckoutButton() {
    return renderDefaultCheckoutButton()
}
```

Fonte: Elaborado pelo autor (2025)

3.5 OUTROS ARQUIVOS

Nem todos os tipos de arquivos ou linguagens suportam comentários para anotações, como é o caso do JSON. Além disso, há cenários nos quais o código é modificado indiretamente por meio de configurações externas ou templates, em vez de edição direta. Exemplos incluem arquivos de configuração gerados dinamicamente por sistemas de build ou mecanismos de templating. Esses casos impõem desafios específicos à gestão da variabilidade, uma vez que métodos baseados em anotações tradicionais não são aplicáveis.

Para tais situações, o FLAG oferece flexibilidade ao gerenciar a variabilidade por meio de um sistema de controle de versões personalizado. Ele mantém uma árvore de versões para cada arquivo, associando modificações a funcionalidades específicas. O FLAG determina dinamicamente qual versão deve ser exibida com base no estado atual das funcionalidades. Além disso, FLAG é capaz de realizar a mesclagem de versões e resolver conflitos, assegurando consistência mesmo quando há sobreposição ou evolução das funcionalidades. Essa capacidade torna o

FLAG altamente adaptável a diferentes tipos de arquivos e fluxos de trabalho, estendendo sua aplicabilidade além de arquivos de código tradicionais.

4 PLANO DE ESTUDO EXPERIMENTAL

Nesta seção, detalhamos o estudo experimental conduzido para avaliar a abordagem proposta. A fim de garantir a confiabilidade dos resultados, seguimos as diretrizes estabelecidas por Juristo e Moreno (JURISTO; MORENO, 2013) e por Wohlin et al. (WOHLIN et al., 2012) para a realização de experimentos na área de engenharia de software.

Adicionalmente, fornecemos todas as informações necessárias para permitir possíveis replicações do nosso estudo, incluindo o plano e os objetivos da pesquisa, hipóteses, tratamentos, objetos de controle, objetos experimentais e variáveis, bem como o planejamento, preparação, execução, análise dos resultados e ameaças à validade.

4.1 OBJETIVO, QUESTÃO E MÉTRICAS (GQM)

Seguindo as diretrizes de Basili (BASILI, 1994), definimos o objetivo, a questão de pesquisa e as métricas utilizadas em nosso estudo.

4.1.1 Objetivo

O objetivo deste experimento é analisar a complexidade de software em projetos de código aberto que utilizam algum mecanismo de variabilidade de software.

4.1.2 Questão

Para atingir esse objetivo, definimos a seguinte questão de pesquisa:

RQ1: O FLAG reduz a complexidade do código quando comparado com abordagens existentes?

4.1.3 Métricas

Para responder à RQ1, é necessário verificar se o uso do FLAG reduz quantitativamente a complexidade do código. Para isso, selecionamos métricas que avaliam a complexidade por meio de análise estática. Diversas métricas são propostas na literatura para medir a complexidade do código (EL-SHARKAWY; YAMAGISHI-EICHLER; SCHMID, 2019), e diferentes autores

justificam a escolha de determinadas métricas em detrimento de outras. Inspirados em Antinyan et al. (ANTINYAN; STARON; SANDBERG, 2017), destacamos três principais fatores que contribuem para o aumento da complexidade: (1) arquitetura inadequada, (2) profundidade de aninhamento e (3) estruturas de controle. Como não estamos lidando com modelagem ou aspectos arquiteturais, selecionamos duas métricas: (1) Complexidade Ciclomática e (2) Complexidade Cognitiva.

- ***Complexidade Ciclomatica* (CiC):** A métrica proposta por McCabe (MCCABE, 1976) foi originalmente desenvolvida para avaliar se um componente ou módulo é testável e passível de manutenção. Essa métrica quantifica a complexidade com base no número de caminhos distintos que o programa pode seguir. Quanto maior a complexidade, mais difícil é manter o código e maior a propensão a erros.

- ***Complexidade Cognitiva* (CoG):** Proposta por Campbell (CAMPBELL, 2018), essa métrica foca na compreensibilidade do código. Semelhante à complexidade ciclomática, ela avalia a profundidade do aninhamento, influenciando diretamente a facilidade com que um desenvolvedor entende e navega na lógica do sistema. Ao enfatizar a carga cognitiva imposta por estruturas de controle intrincadas, essa métrica oferece uma visão mais refinada da legibilidade e manutenibilidade do código.

Outras métricas poderiam ser utilizadas, como Linhas Lógicas de Código, Profundidade de Aninhamento, Índice de Manutenibilidade (COLEMAN et al., 1994), e Volume de Halstead (HALSTEAD, 1977). Essas métricas muitas vezes avaliam aspectos similares ou estão correlacionadas com a contagem de linhas de código. Além disso, métricas como a CiC são frequentemente utilizadas como base para o cálculo de outras métricas (EL-SHARKAWY; YAMAGISHI-EICHLER; SCHMID, 2019; GARG, 2014).

Ambas as métricas foram obtidas por meio de análise estática com a ferramenta SonarQube[1], que oferece um ambiente de medição integrado à nossa *Integrated Development Environment* (IDE), no caso, o Visual Studio Code[2].

[1] https://www.sonarsource.com/
[2] https://code.visualstudio.com/

4.2 PLANEJAMENTO

Com base no objetivo, nas questões e nas métricas definidas, apresentamos nesta seção informações adicionais sobre o planejamento do experimento, incluindo as hipóteses, os tratamentos e as variáveis dependentes e independentes.

4.2.1 Definição das Hipóteses

Para responder à RQ1, definimos a hipótese nula e a hipótese alternativa. A hipótese nula postula que não há diferença estatisticamente significativa na Complexidade Cognitiva e Ciclomática entre o código antes e depois da aplicação do FLAG. Por sua vez, a hipótese alternativa, aceita caso a hipótese nula seja rejeitada, indica que há uma redução significativa na complexidade do código (CoG e CiC) após a aplicação do FLAG.

Hipótese Nula ($H0_1$). Não há diferença estatisticamente significativa na complexidade do código (CoG e CiC) entre o código antes e depois da aplicação do FLAG.

Hipótese Alternativa ($H1_1$). A aplicação do FLAG reduz significativamente a complexidade do código (CoG e CiC).

4.2.2 Tratamento

O tratamento aplicado neste estudo é o uso da ferramenta FLAG. Trata-se de um experimento com um único fator, pois estamos investigando exclusivamente o impacto da ferramenta. A condição de controle, ou seja, a ausência do tratamento, consiste na medição da complexidade do código (CoG e CiC) nos métodos e funções originais, antes da aplicação do FLAG.

$$H_0 : \mu_{\text{depois}} \geq \mu_{\text{antes}} \quad \text{(onde } \mu \text{ representa a mediana de CoG e CiC)}$$
$$H_1 : \mu_{\text{depois}} < \mu_{\text{antes}}$$

4.2.3 Objetos de Controle

Os objetos de controle são os métodos e funções originais, extraídos de 17 projetos de código aberto escritos em C/C++, Java, Python, Go e C#, antes da aplicação da ferra-

menta. Esses objetos representam a complexidade de base do código e servem como ponto de referência para comparação.

4.2.4 Objetos Experimentais

Os objetos experimentais consistem nos mesmos métodos e funções utilizados como controle, mas modificados com a aplicação do FLAG. Esses objetos representam o código após o tratamento, permitindo avaliar o impacto da ferramenta.

4.2.5 Variáveis Independentes

A variável independente deste estudo é a aplicação da ferramenta FLAG, apresentada como uma variável binária, com dois estados distintos: 'FLAG aplicado' (grupo experimental) e 'FLAG não aplicado' (grupo de controle). Incluímos também as ferramentas utilizadas para coleta e análise dos dados.

4.2.6 Variáveis Dependentes

As variáveis dependentes são a Complexidade Cognitiva (CoG) e a Complexidade Ciclomática (CiC) dos métodos e funções analisados.

4.2.7 Desenho dos Ensaios

Para avaliar o impacto da ferramenta na complexidade do código, aplicamos o FLAG em métodos e funções selecionados de projetos open-source. Medimos os valores de CoG e CiC antes e depois da aplicação da ferramenta. Em seguida, comparamos os resultados para identificar possíveis reduções de complexidade. Por fim, aplicamos testes estatísticos para verificar a significância das mudanças observadas. A Seção 5 apresenta os procedimentos e a análise dos dados em detalhes.

4.3 PREPARAÇÃO

Selecionamos 17 projetos de código aberto, em diferentes linguagens, para realizar a avaliação. O critério de seleção adotado foi híbrido: priorizamos projetos bem conhecidos pela comunidade, além de empregar uma metodologia — detalhada a seguir — para identificar no GitHub repositórios que utilizam alguma técnica de variabilidade de código. A Tabela 2 lista os projetos utilizados e suas respectivas linguagens.

Tabela 2 – Repositórios utilizados.

Repositório	**Linguagem**
libssh/libssh-mirror	C/C++
keycloak/keycloak	Java
readthedocs/readthedocs.org	Python
openedx-unsupported/edx-analytics-dashboard	Python
openedx/course-discovery	Python
gustaf-a/BookManage	Python
grafana/grafana	Golang
sillsdev/web-xforge	C#
richardforrestbarker/Bardcoded	C#
Altinn/altinn-authentication	C#
LovassyApp/LovassyApp	C#
JaCraig/Mithril	C#
OperationFman/budget-sherpa	C#
Theodor-Springmann-Stiftung/hamann-ausgabe-core	C#
DigiBanks99/menlo	C#
aseriousbiz/abbot-web	C#

Fonte: Elaborada pelo autor (2025)

Selecionamos os projetos Libssh, Keycloak e Grafana por sua relevância no ecossistema de software livre. Inspirados na metodologia de Hoyos et al. (HOYOS et al., 2021), selecionamos os demais projetos conforme os seguintes critérios:

- Utilização da API de busca do GitHub com a seguinte string de pesquisa: "<biblioteca> language:<linguagem>", adaptando com sinônimos quando necessário, por exemplo, "django-waffle language: Python".
- Exclusão de repositórios sem descrição.
- Exclusão de forks.

- Exclusão de repositórios contendo os termos “demo”, “test”, “PoC” ou similares.

Alguns fatores limitaram o número total de repositórios analisados. Além da ausência de suporte do SonarQube para análise da complexidade ciclomática (CiC) em Go — o que exigiu o uso da ferramenta Codalyze (Selçuk Usta, 2025) —, outros três fatores contribuíram: (1) o esforço manual necessário para seleção e coleta de dados, conforme discutido na Seção 4.3.1; (2) a necessidade de implementar um script personalizado para o cálculo das métricas em C/C++, detalhada na Seção 4.3.2; e (3) a exclusão de projetos JavaScript devido ao seu padrão estrutural baseado em funções aninhadas e callbacks, que dificultava a mensuração objetiva da complexidade. Apesar dessas limitações, a amostra coletada é suficientemente representativa para fins experimentais.

4.3.1 Operação

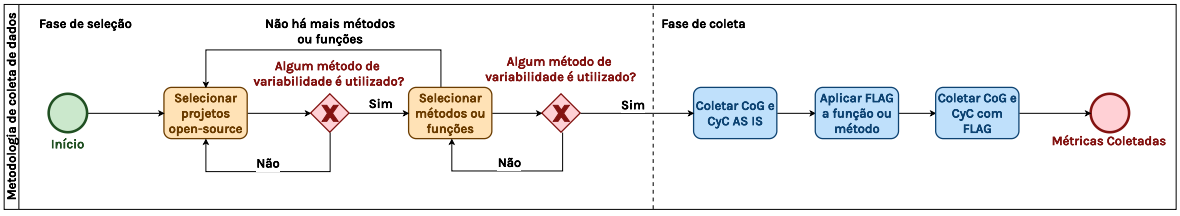
Para conduzir o experimento, inicialmente identificamos os pontos de variabilidade no código, ou seja, trechos em que estruturas condicionais ou diretivas de pré-processamento controlavam comportamentos distintos. Em seguida, calculamos as métricas de complexidade no código original, não modificado, a fim de obter uma linha de base. Após essa análise inicial, modificamos o código para utilizar o FLAG, substituindo as estruturas tradicionais por sua abordagem de gerenciamento de variabilidade, e realizamos uma nova medição de complexidade. Por fim, colocamos o bloco criado em modo ativo — ou seja, fora do modo de desenvolvimento — e coletamos novamente as métricas com essa versão, simulando o período em que o bloco estaria efetivamente ativado na base de código.

Inicialmente, houve uma tentativa de automatizar esse processo; contudo, da mesma forma que identificar funcionalidades de forma programática é um desafio, automatizar esse processo também se mostrou bastante difícil, o que nos levou a descontinuar essa abordagem. Os resultados obtidos foram separados por linguagem para análise posterior.

Enquanto modificávamos o código-fonte, buscamos substituir estruturas tradicionais de controle, como comandos `if/else` e diretivas de pré-processamento, pelas instruções fornecidas pelo FLAG. Ao medir a complexidade, tivemos algumas opções: calcular as métricas no nível de módulos, componentes, arquivos ou funções/métodos. Optamos pelo nível de função, pois essa abordagem fornece uma forma consistente e granular de avaliar a complexidade, facilitando a comparação entre diferentes linguagens de programação.

Adotamos uma estratégia conservadora na fase de coleta dos resultados para garantir a precisão das análises. Um caso notável ocorreu em funções ou métodos que continham pontos de variabilidade — seja por meio de pré-processadores ou estruturas condicionais — em que o número de linhas do ramo `if` era significativamente maior do que o do ramo `else`, ou vice-versa. Nesses casos, optamos por basear nossa análise na versão mais extensa do código, a fim de evitar a sub-representação da sua complexidade. Além disso, em algumas situações — principalmente com pré-processadores — a simples substituição da instrução de variabilidade comprometia o funcionamento do código. A correção desses problemas exigia modificações adicionais, o que poderia introduzir viés nos resultados. Para manter a integridade do estudo, decidimos descartar esses métodos da análise, assegurando que nossos achados permanecessem confiáveis e imparciais. Na figura 2 podemos ver diagrama consolidando a abordagem de coleta de dados para o experimento.

Figura 2 – Visão geral da metodologia de coleta de dados



Fonte: Elaborada pelo autor (2025)

4.3.2 Limitações do Sonar para C++

Planejamos a avaliação de projetos C++ com o intuito de comparar o FLAG com o uso de pré-processadores. No entanto, surgiu um problema significativo durante a coleta dos resultados: o SonarQube não computava a complexidade de trechos de código controlados por diretivas de pré-processamento. Posteriormente, a equipe de suporte do SonarQube confirmou que essa é, de fato, uma limitação da ferramenta, pois, internamente, ela utiliza *Abstract syntax trees* (ASTs) para realizar os cálculos (SonarCommunity, 2025).

Considerando essa limitação, desenvolvemos um script em Golang para calcular as métricas de CiC e CoG, tratando as diretivas de pré-processamento como estruturas condicionais. Esse script está disponível para replicação.

4.4 ANÁLISE

A análise se concentra na avaliação da eficácia do FLAG na redução da complexidade do código. Especificamente, comparamos as métricas CoG e CiC medidas nas funções e métodos selecionados antes (objetos de controle) e depois (objetos experimentais) da aplicação do FLAG.

Utilizamos o teste de postos sinalizados de Wilcoxon para analisar os resultados, considerando o estudo como unifatorial com dados que não seguem uma distribuição normal. O teste de Wilcoxon é um método estatístico não paramétrico utilizado para verificar se duas amostras emparelhadas provém da mesma distribuição, sendo apropriado para comparar amostras de tamanhos iguais ou diferentes (WOOLSON, 2005).

4.5 AMEAÇAS À VALIDADE

Esta seção discute a validade dos resultados obtidos e a possibilidade de generalizá-los para uma população mais ampla.

- **Validade Interna.** Uma ameaça observável está no uso de um script personalizado para C++ e da ferramenta Codalyze para Golang, o que introduz o risco de inconsistências na forma como as métricas de complexidade são calculadas em comparação com o SonarQube. Isso pode gerar resultados enviesados, uma vez que os instrumentos de medição não são uniformes. Além disso, há o problema de termos analisado um número reduzido de projetos e coletado uma quantidade limitada de dados. Para mitigar essa limitação, optamos por projetos variados, o que ajudou a cobrir diferentes cenários de desenvolvimento.

- **Validade Externa.** Outro ponto a considerar é que, embora tenhamos validado o estudo por meio da análise de diferentes repositórios, isso não constitui uma prova definitiva de correção. Em um caso real, se os desenvolvedores utilizarem nossa ferramenta desde o início do desenvolvimento, isso poderá alterar a forma como escrevem o código, o que poderia impactar os resultados observados. Ademais, o número restrito de projetos incluídos neste experimento compromete a validade externa dos nossos achados, especialmente sua generalização. Embora tenhamos selecionado estrategicamente projetos variados para representar diferentes contextos de desenvolvimento, o tamanho redu-

zido da amostra pode não representar adequadamente o universo de projetos de código aberto. Portanto, nossos resultados só podem ser extrapolados para outros projetos e ambientes de desenvolvimento com certa cautela.

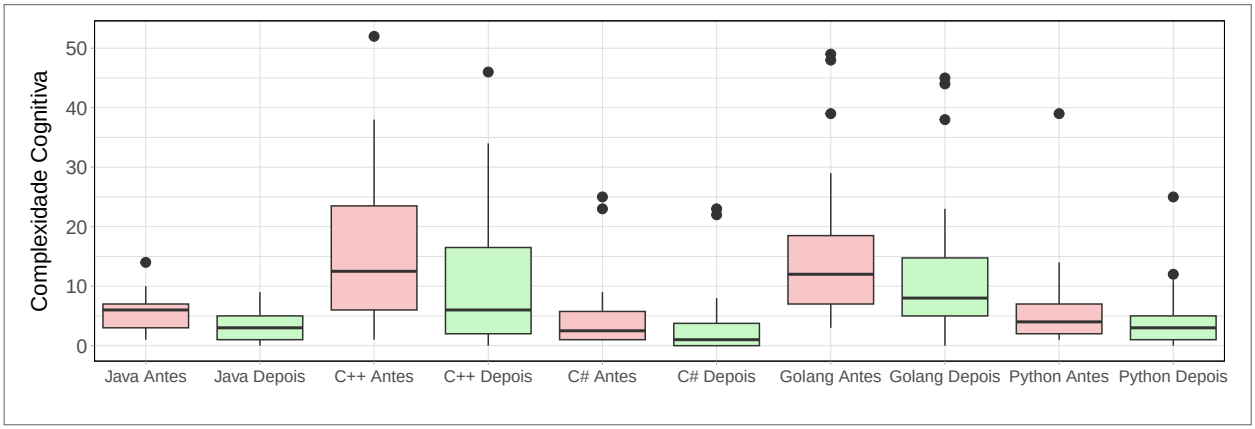
5 RESULTADOS

Neste capítulo apresentaremos e discutiremos os resultados obtidos a partir dos dados coletados por meio da metodologia descrita na seção anterior. Com o objetivo de viabilizar a verificação independente dos resultados, disponibilizamos os artefatos produzidos para a realização deste estudo[1]. Nosso material suplementar (Lucas Costa, 2025) inclui a implementação dos scripts para cálculo das métricas, os dados brutos e os scripts utilizados para a análise estatística. Ao final desta seção, realizaremos uma análise global dos dados, integrando os resultados de todas as linguagens a fim de evidenciar padrões e tendências mais abrangentes.

A distribuição das complexidades Cognitiva e Ciclomática para as linguagens testadas é apresentada no formato de boxplot nas Figuras 4 e 5, respectivamente. As subseções a seguir discutirão os resultados de cada métrica com maior detalhamento.

5.1 COMPLEXIDADE COGNITIVA

Figura 3 – Resultados de Complexidade Cognitiva Antes e Depois da Aplicação do FLAG

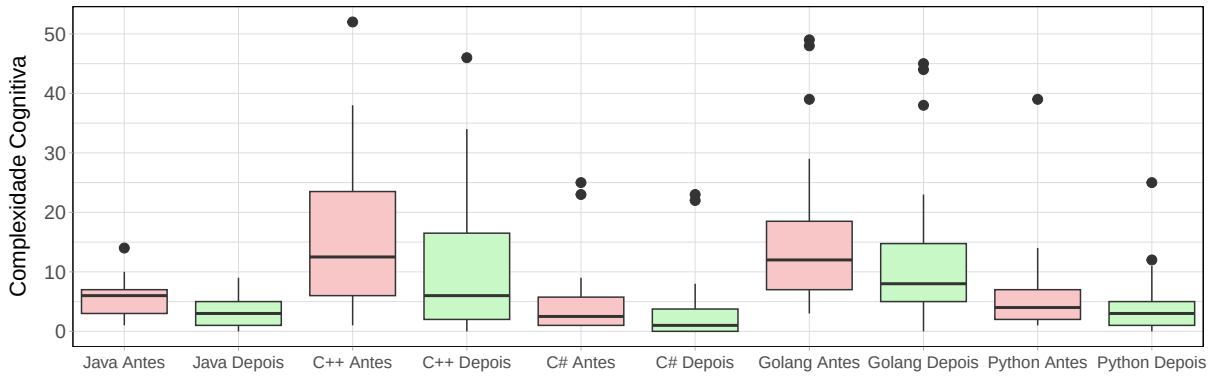


Fonte: Elaborada pelo autor (2025)

A análise visual dos dados evidencia que FLAG, como esperado, reduziu todas as medianas e médias para todas as linguagens testadas, indicando uma mudança na tendência central dos dados. Com relação aos *Interquartile Range* (IQR) apresentados na Tabela 3, a linguagem C/C++ apresentou os melhores resultados, com uma redução de 3 pontos no IQR, seguida por Golang (1,75), Python (1), C# (1) e Java (0). Também é possível observar que os "whiskers" se retraíram, assim como a quantidade de outliers nas distribuições.

[1] https://github.com/costaluu/submission

Figura 4 – Resultados de Complexidade Cognitiva Antes e Depois da Aplicação do FLAG



Fonte: Elaborada pelo autor (2025)

Tabela 3 – Comparação de CoG de métricas antes e depois para diferentes linguagens.

Linguagem	**Complexidade Cognitiva**						**Redução**		
	Mediana		**Média**		**IQR**		**Abs.**	**Rel.**	**DP**
	Antes	Depois	Antes	Depois	Antes	Depois			
C/C++	12.5	6	16.2	10.7	17.5	14.5	-5.5	33.9%	5.41
Golang	12	8	15.4	12.1	11.5	9.75	-3.9	21.4%	3.43
Java	6	3	5.23	3.27	4	4	-1.96	37.4%	1.47
Python	4	3	6	4.03	5	4	-1.97	32.8%	2.74
C#	2.5	1	4.67	3.37	4.75	3.75	-1.3	27.8%	1.02

DP: Desvio Padrão. **Abs.**: Redução absoluta. **Rel.**: Redução Relativa.

Fonte: Elaborada pelo autor (2025)

A análise do IQR revela apenas a dispersão dos dados antes e depois da aplicação do FLAG. A redução média absoluta variou entre 5,5 e 1,3, indicando uma diminuição consistente (acima de 1) na Complexidade Cognitiva por função ou método. Entre as linguagens analisadas, C++ destacou-se por apresentar a maior redução absoluta (5,5, com redução relativa de 33,9%), seguida por Golang (3,9, 21,4%), Java (1,97, 37,6%), Python (1,97, 32,8%) e C# (1,3, 27,8%). O desvio padrão apresentou variações proporcionais à magnitude das reduções, sendo maior nas regiões com maior redução e menor onde a redução foi menos expressiva.

Após a avaliação visual, prosseguimos com a análise estatística dos dados. Como não foi possível assumir normalidade na distribuição dos dados[2], utilizamos o teste de postos sinalizados de Wilcoxon, com nível de significância de 5%; a hipótese nula considera que a diferença das medianas das observações pareadas é zero. As diferenças observadas na Complexidade

[2] O teste de normalidade de Shapiro-Wilk (SHAPIRO; WILK, 1965) confirmou que a distribuição não é normal.

Cognitiva foram estatisticamente significativas para todas as linguagens testadas, conforme demonstrado na Tabela 4.

Os resultados apresentados na Tabela 4 fornecem uma visão detalhada das diferenças entre linguagens de programação com base na análise estatística utilizando o teste de Wilcoxon e a correlação bisserial de postos pareados. Os valores de p são altamente significativos (todos inferiores a $5e-6$), o que indica forte evidência de diferenças nas métricas avaliadas. Os tamanhos de efeito variam de 0,873 a 0,924, sugerindo um efeito forte e consistente entre as linguagens. Dentre elas, a linguagem Python apresentou o maior tamanho de efeito (0,924), seguida por C# (0,906) e Java (0,894), sugerindo que essas linguagens demonstram as maiores diferenças na métrica analisada.

Tabela 4 – Resultados estatísticos para CoG em diferentes linguagens de programação.

Linguagem	V	$p-value$	Tamanho do efeito	Intervalo de Confiança
C/C++	435	2.543e-06	0.873	86-88%
Golang	435	2.399e-06	0.875	87-88%
Java	465	1.039e-06	0.894	88-92%
Python	465	4.386e-07	0.924	90-96%
C#	406	1.137e-06	0.906	87-95%

Tamanho do sample por linguagem: 30. **V**: Teste estatistico Wilcoxon Signed-Rank. **Tamanho do efeito**: Matched-pairs rank-biserial correlation.

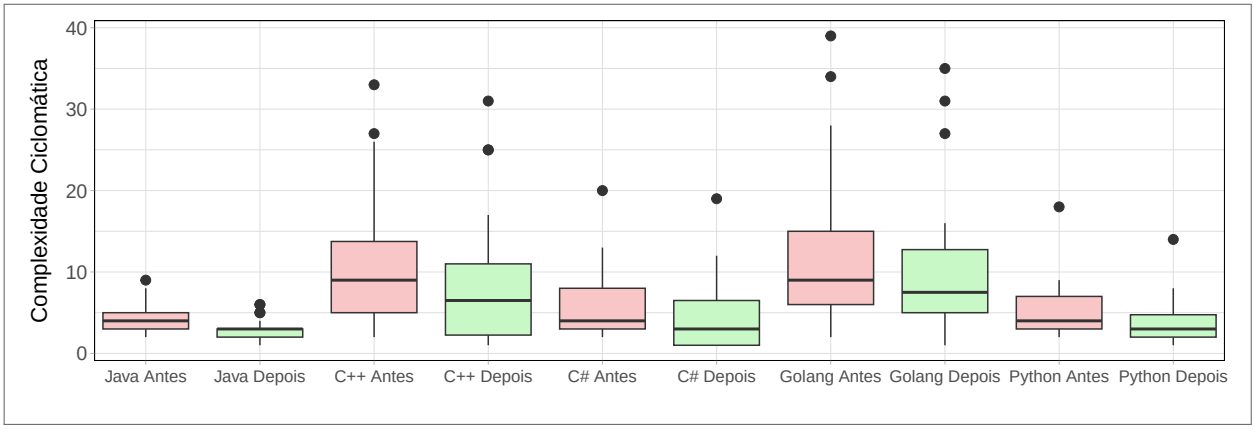
Fonte: Elaborada pelo autor (2025)

Os intervalos de confiança (ICs) para os testes de Wilcoxon e para a correlação bisserial, em alguns casos, ficaram abaixo do limiar convencional de 95%. Isso se deve ao tamanho reduzido da amostra e ao fato de que, diferentemente dos testes paramétricos que utilizam distribuições contínuas, o teste de Wilcoxon opera sobre postos discretos, o que pode gerar irregularidades no cálculo do intervalo de confiança, especialmente quando o número de observações é pequeno.

5.2 COMPLEXIDADE CICLOMÁTICA

Conforme observamos na análise da Complexidade Cognitiva, a avaliação visual dos dados também confirma que o FLAG reduziu consistentemente os valores medianos e médios em todas as linguagens testadas, indicando uma mudança na tendência central dos dados. Em relação aos intervalos interquartis (IQRs) apresentados na Tabela 5, a linguagem Golang apresentou a maior redução, com uma diminuição de 1.75 pontos no IQR, seguida por Python (1.25),

Figura 5 – Boxplot mostrando mudanças na Complexidade Ciclomática na Complexidade Ciclomática Antes e Depois da Aplicação do FLAG



Fonte: Elaborada pelo autor (2025)

Java (1), C/C++ (0) e C# com um aumento de 0.5 pontos. Além disso, os limites extremos (whiskers) das distribuições encolheram e o número de outliers diminuiu, reforçando ainda mais a redução observada na complexidade.

Tabela 5 – Comparação de CiC de métricas antes e depois para diferentes linguagens.

Linguagem	**Complexidade Ciclomática**						**Redução**		
	Mediana		**Média**		**IQR**		**Abs.**	**Rel.**	**DP**
	Antes	Depois	Antes	Depois	Antes	Depois			
C/C++	9	6.5	10.7	8.47	8.75	8.75	-2.23	20.8%	1.68
Golang	9	7.5	12.1	10.1	9	7.75	-2	16.5%	2.2
Java	4	3	4	2.7	2	1	-1.3	32.5%	0.837
Python	4	3	5	3.77	4	2.75	-1.23	24.7%	0.858
C#	4	3	5.73	4.5	5	5.5	-1.23	21.5%	0.774

DP: Desvio Padrão. **Abs.**: Redução absoluta. **Rel.**: Redução Relativa.

Fonte: Elaborada pelo autor (2025)

A redução absoluta média variou entre 2.3 e 1.23, indicando uma redução consistente (superior a 1) na Complexidade Ciclomática por função. Entre as linguagens, C++ destacou-se por apresentar a maior redução absoluta (2.23, redução relativa de 20.8%), seguida por Golang (2, redução relativa de 16.5%), Java (1.3, 32.5%), Python (1.23, 24.7%) e C# (1.23, 21.5%). A variação padrão, assim como na Complexidade Cognitiva, também variou proporcionalmente à magnitude das reduções, sendo maior nas regiões com maiores reduções e menor onde a redução foi menos significativa.

Após a avaliação visual, procedemos com a análise estatística dos dados. Reexecutamos

o teste de normalidade e confirmamos que as distribuições não seguem uma distribuição normal. Com os parâmetros anteriores, utilizamos o teste de postos sinalizados de Wilcoxon; a hipótese nula assume que a diferença mediana entre as observações pareadas é zero. As diferenças observadas na CiC foram estatisticamente significativas para todas as linguagens testadas, conforme apresentado na Tabela 6.

Tabela 6 – Resultados estatísticos para o CiC em diferentes linguagens de programação.

Linguagem	V	$p - value$	Tamanho do Efeito	Intervalo de Confiança
C/C++	465	1.243e-06	0.887	88-91%
Java	465	2.765e-07	0.940	91-99%
Python	435	2.556e-07	0.948	91-99%
Golang	435	1.459e-06	0.892	87-92%
C# 30	465	2.094e-07	0.950	92-99%

Tamanho do sample por linguagem: 30. **V**: Teste estatistico Wilcoxon Signed-Rank. **Tamanho do efeito**: Matched-pairs rank-biserial correlation.

Fonte: Elaborada pelo autor (2025)

Uma análise mais aprofundada dos resultados na Tabela 6 revela que os valores de p em todas as linguagens são altamente significativos (todos abaixo de $2e-6$), indicando fortes evidências de diferenças na métrica avaliada. Os tamanhos do efeito variam entre 0.887 e 0.950, sugerindo um efeito fortemente consistente entre as linguagens. Dentro desse grupo, C# apresenta o maior tamanho do efeito (0.950), seguido por Python (0.948) e Java (0.940), o que sugere que essas linguagens demonstram a maior diferença no atributo medido.

Resposta à RQ1. Nossos resultados demonstram que o FLAG reduz significativamente a complexidade cognitiva e ciclomática. Em média, foram reduzidos aproximadamente 28 pontos de complexidade cognitiva para cada 10 funções ou métodos tratados com o FLAG. Além disso, a complexidade ciclomática foi reduzida em cerca de 16 pontos para cada 10 funções ou métodos. Em termos práticos, o FLAG removeu cerca de 16 instruções condicionais (ou instruções de pré-processadores) para cada 10 funções ou métodos analisados.

6 CONCLUSÃO

Este artigo apresentou o FLAG, uma abordagem independente de linguagem para o gerenciamento de variabilidade, projetada com o objetivo de reduzir a complexidade de software em Linhas de Produto de Software (SPLs). A complexidade estrutural e a elevada interdependência entre variantes de produto constituem desafios fundamentais no desenvolvimento de SPLs, impactando diretamente a manutenção e a evolução dos sistemas. O FLAG busca mitigar esses desafios ao fornecer uma estratégia sistemática para o controle da variabilidade, reduzindo o esforço de manutenção e contribuindo para a diminuição da dívida técnica.

Os experimentos realizados em 17 projetos de código aberto, abrangendo cinco diferentes linguagens de programação, demonstraram que o FLAG reduziu a complexidade de forma estatisticamente significativa em todos os cenários analisados. Esses resultados indicam que o FLAG pode ser uma alternativa promissora para o gerenciamento de variabilidade em SPLs, favorecendo a modularidade, a manutenibilidade e a escalabilidade de famílias de produtos de software.

Como parte dos trabalhos futuros, pretende-se ampliar o escopo dos experimentos para incluir um conjunto mais amplo de linguagens de programação e projetos de código aberto. Um número maior de projetos e linguagens permitirá uma avaliação mais abrangente e precisa do impacto do FLAG na redução da complexidade em diferentes contextos de desenvolvimento. Pretendemos também analisar os efeitos do FLAG ao longo de todo o ciclo de vida do software, examinando sua influência em atividades como refatoração, evolução do sistema e custos de manutenção. Essas investigações contribuirão para consolidar o FLAG como uma solução eficaz para o gerenciamento de variabilidade e a redução da complexidade no desenvolvimento de software.

REFERÊNCIAS

AMIN, A. K. M. Fazal-e; OXLEY, A. A review on aspect oriented implementation of software product lines components. *Information Technology Journal*, v. 9, n. 6, p. 1262–1269, 2010.

ANTINYAN, V.; STARON, M.; SANDBERG, A. Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. *Empirical Software Engineering*, Springer, v. 22, p. 3057–3087, 2017.

BACHMANN, F.; CLEMENTS, P. *Variability in software product lines*. [S.l.]: Carnegie Mellon University, Software Engineering Institute, 2005.

BASILI, V. R. Goal, question, metric paradigm. *Encyclopedia of software engineering*, John Wiley & Sons, v. 1, p. 528–532, 1994.

BASTOS, J. F.; NETO, P. A. d. M. S.; O'LEARY, P.; ALMEIDA, E. S. de; MEIRA, S. R. de L. Software product lines adoption in small organizations. *Journal of Systems and Software*, Elsevier, v. 131, p. 112–128, 2017.

BENAROCH, M.; LYYTINEN, K. How much does software complexity matter for maintenance productivity? the link between team instability and diversity. *IEEE Transactions on Software Engineering*, IEEE, v. 49, n. 4, p. 2459–2475, 2022.

BLACK, A. P. Object-oriented programming: Some history, and challenges for the next fifty years. *Information and Computation*, Elsevier, v. 231, p. 3–20, 2013.

BREIVOLD, H. P.; LARSSON, S.; LAND, R. Migrating industrial systems towards software product lines: Experiences and observations through case studies. In: IEEE. *2008 34th Euromicro Conference Software Engineering and Advanced Applications*. [S.l.], 2008. p. 232–239.

CAMPBELL, G. A. Cognitive complexity: an overview and evaluation. In: *Proceedings of the 2018 International Conference on Technical Debt*. New York, NY, USA: Association for Computing Machinery, 2018. (TechDebt '18), p. 57–58. ISBN 9781450357135. Disponível em: <https://doi.org/10.1145/3194164.3194186>.

CHEN, L.; BABAR, M. A. A systematic review of evaluation of variability management approaches in software product lines. *Information and Software Technology*, Elsevier, v. 53, n. 4, p. 344–362, 2011.

CLEMENTS, P.; NORTHROP, L. *Software product lines*. [S.l.]: Addison-Wesley Boston, 2002. v. 1.

COHEN, M. B.; DWYER, M. B.; SHI, J. Coverage and adequacy in software product line testing. In: *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*. [S.l.: s.n.], 2006. p. 53–63.

COLEMAN, D.; ASH, D.; LOWTHER, B.; OMAN, P. Using metrics to evaluate software system maintainability. *Computer*, IEEE, v. 27, n. 8, p. 44–49, 1994.

DANIEL, A. O.; OMOBOLANLE, O. O. Economic impact of software product line engineering method–a survey. *International Journal of Advanced Research in Computer Science*, v. 11, n. 5, 2020.

EL-SHARKAWY, S.; YAMAGISHI-EICHLER, N.; SCHMID, K. Metrics for analyzing variability and its implementation in software product lines: A systematic literature review. *Information and Software Technology*, Elsevier, v. 106, p. 1–30, 2019.

FEITELSON, D. G.; FRACHTENBERG, E.; BECK, K. L. Development and deployment at facebook. *IEEE Internet Computing*, IEEE, v. 17, n. 4, p. 8–17, 2013.

GARG, A. An approach for improving the concept of cyclomatic complexity for object-oriented programming. *arXiv preprint arXiv:1412.6216*, 2014.

HALSTEAD, M. H. *Elements of Software Science (Operating and programming systems series)*. [S.l.]: Elsevier Science Inc., 1977.

HOYOS, J.; ABDALKAREEM, R.; MUJAHID, S.; SHIHAB, E.; BEDOYA, A. E. On the removal of feature toggles: A study of python projects and practitioners motivations. *Empirical Software Engineering*, Springer, v. 26, p. 1–26, 2021.

JÉZÉQUEL, J.-M.; KIENZLE, J.; ACHER, M. From feature models to feature toggles in practice. In: *Proceedings of the 26th ACM International Systems and Software Product Line Conference-Volume A*. [S.l.: s.n.], 2022. p. 234–244.

JURISTO, N.; MORENO, A. M. *Basics of software engineering experimentation*. [S.l.]: Springer Science & Business Media, 2013.

KRÜGER, J.; BERGER, T.; LEICH, T. Features and how to find them: a survey of manual feature location. In: *Software Engineering for Variability Intensive Systems*. [S.l.]: Auerbach Publications, 2019. p. 153–172.

LIEBIG, J.; APEL, S.; LENGAUER, C.; KÄSTNER, C.; SCHULZE, M. An analysis of the variability in forty preprocessor-based software product lines. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. [S.l.: s.n.], 2010. p. 105–114.

LINDEN, F. J. Van der; SCHMID, K.; ROMMES, E. *Software product lines in action: the best industrial practice in product line engineering*. [S.l.]: Springer Science & Business Media, 2007.

LOPES, B.; AMORIM, S.; FERREIRA, C. Solution discovery over feature toggling with built-in abstraction in outsystems. In: *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. [S.l.: s.n.], 2021. p. 47–56.

Lucas Costa, B. *Supplementary material: Minimizing Code Complexity and Enhancing Maintainability Through Language-Agnostic Variability Management*. 2025. <https://github.com/costaluu/submission>. Accessed on: March 31, 2025.

MAHDAVI-HEZAVEH, R.; FATIMA, S.; WILLIAMS, L. Paving a path for a combined family of feature toggle and configuration option research. *ACM Transactions on Software Engineering and Methodology*, ACM New York, NY, v. 33, n. 7, p. 1–27, 2024.

MAHMOOD, A. K.; OXLEY, A. An analysis of object oriented variability implementation mechanisms. *ACM SIGSOFT Software Engineering Notes*, ACM New York, NY, USA, v. 36, n. 1, p. 1–4, 2011.

MCCABE, T. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2, n. 4, p. 308–320, 1976.

MEDEIROS, F.; KÄSTNER, C.; RIBEIRO, M.; NADI, S.; GHEYI, R. The love/hate relationship with the c preprocessor: An interview study. In: SCHLOSS DAGSTUHL–LEIBNIZ-ZENTRUM FÜR INFORMATIK. *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. [S.l.], 2015. p. 495–518.

MEDEIROS, F.; RIBEIRO, M.; GHEYI, R.; APEL, S.; KÄSTNER, C.; FERREIRA, B.; CARVALHO, L.; FONSECA, B. Discipline matters: Refactoring of preprocessor directives in the# ifdef hell. *IEEE Transactions on Software Engineering*, IEEE, v. 44, n. 5, p. 453–469, 2017.

MEINICKE, J.; WONG, C.-P.; VASILESCU, B.; KäSTNER, C. Exploring differences and commonalities between feature flags and configuration options. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. New York, NY, USA: Association for Computing Machinery, 2020. (ICSE-SEIP '20), p. 233–242. ISBN 9781450371230. Disponível em: <https://doi.org/10.1145/3377813.3381366>.

METZGER, A.; POHL, K. Software product line engineering and variability management: achievements and challenges. *Future of software engineering proceedings*, p. 70–84, 2014.

MIDHA, V. Does complexity matter? the impact of change in structural complexity on software maintenance and new developers' contributions in open source software. 2008.

NORTHROP, L.; CLEMENTS, P.; BACHMANN, F.; BERGEY, J.; CHASTEK, G.; COHEN, S.; DONOHOE, P.; JONES, L.; KRUT, R.; LITTLE, R. et al. A framework for software product line practice, version 5.0. *SEI.–2007–http://www. sei. cmu. edu/productlines/index. html*, 2007.

OGHENEOVO, E. E. et al. On the relationship between software complexity and maintenance costs. *Journal of Computer and Communications*, Scientific Research Publishing, v. 2, n. 14, p. 1, 2014.

POHL, K.; BÖCKLE, G.; LINDEN, F. V. D. *Software product line engineering: foundations, principles, and techniques*. [S.l.]: Springer, 2005. v. 1.

PR Newswire. *Feature Flags Have Gone Mainstream, Yet Challenges Remain, Reveals Study From Rollout And Atlassian*. 2020. <https://www.prnewswire.com/news-releases/feature-flags-have-gone-mainstream-yet-challenges-remain-reveals-study-from-rollout-and-atlassian-3007html>. Accessed on: March 23, 2025.

RAHMAN, M. T.; QUEREL, L.-P.; RIGBY, P. C.; ADAMS, B. Feature toggles: practitioner practices and a case study. In: *Proceedings of the 13th international conference on mining software repositories*. [S.l.: s.n.], 2016. p. 201–211.

RAHMAN, T. Feature toggle usage patterns: A case study on google chromium. In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. [S.l.: s.n.], 2023. p. 142–147.

RAMANATHAN, M. K.; CLAPP, L.; BARIK, R.; SRIDHARAN, M. Piranha: Reducing feature flag debt at uber. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. [S.l.: s.n.], 2020. p. 221–230.

RUBIN, J.; CHECHIK, M. A survey of feature location techniques. *Domain Engineering: Product Lines, Languages, and Conceptual Models*, Springer, p. 29–58, 2013.

SCHMID, K.; JOHN, I. A customizable approach to full lifecycle variability management. *Science of Computer Programming*, Elsevier, v. 53, n. 3, p. 259–284, 2004.

Selçuk Usta. *Codalyze - Code Complexity Report Generator*. 2025. <https://marketplace.visualstudio.com/items?itemName=selcuk-usta.code-complexity-report-generator>. Accessed on: March 25, 2025.

SHAPIRO, S. S.; WILK, M. B. An analysis of variance test for normality (complete samples). *Biometrika*, Oxford University Press, v. 52, n. 3-4, p. 591–611, 1965.

SHARVIT, Y. *Data-oriented Programming: Reduce Software Complexity*. [S.l.]: Simon and Schuster, 2022.

SINGH, N.; CHOUHAN, S. S.; VERMA, K. Object oriented programming: Concepts, limitations and application trends. In: IEEE. *2021 5th International Conference on Information Systems and Computer Networks (ISCON)*. [S.l.], 2021. p. 1–4.

SonarCommunity. *Complexity Metrics not being applied to if ifdefs*. 2025. <https://community.sonarsource.com/t/complexity-metrics-not-being-applied-to-if-ifdefs/134677>. Accessed on: March 25, 2025.

SOUZA, L. Oliveria de; ALMEIDA, E. Santana de; NETO, P. A. d. M. S.; BARR, E. T.; PETKE, J. Software product line engineering via software transplantation. *ACM Trans. Softw. Eng. Methodol.*, Association for Computing Machinery, New York, NY, USA, set. 2024. ISSN 1049-331X. Just Accepted. Disponível em: <https://doi.org/10.1145/3695987>.

TARTLER, R.; DIETRICH, C.; SINCERO, J.; SCHRÖDER-PREIKSCHAT, W.; LOHMANN, D. Static analysis of variability in system software: The 90,000# ifdefs issue. In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. [S.l.: s.n.], 2014. p. 421–432.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. et al. *Experimentation in software engineering*. [S.l.]: Springer, 2012. v. 236.

WOOLSON, R. F. Wilcoxon signed-rank test. *Encyclopedia of biostatistics*, Wiley Online Library, v. 8, 2005.