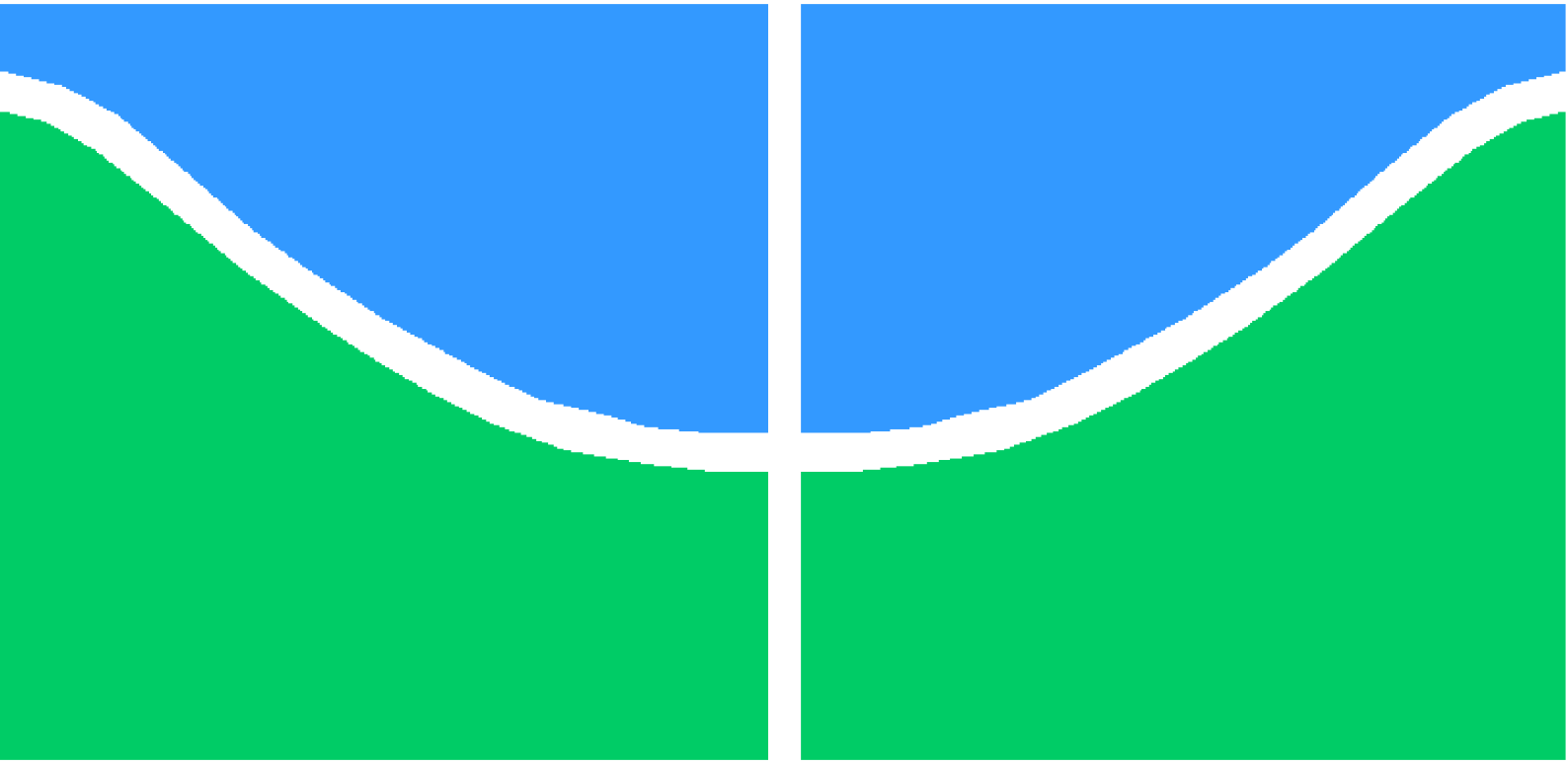


Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Engenharia de Software

Utilização de operações de refatoração para aprimoramento de variabilidade em sistemas de software

Autor: Ronyell Henrique dos Santos

Orientador: Dr. André Luiz Peron Martins Lanna

Brasília, DF

2020

Ronyell Henrique dos Santos

Utilização de operações de refatoração para aprimoramento de variabilidade em sistemas de software

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Dr. André Luiz Peron Martins Lanna

Brasília, DF

2020

Ronyell Henrique dos Santos
Utilização de operações de refatoração para aprimoramento de variabilidade em sistemas de software/ Ronyell Henrique dos Santos. – Brasília, DF, 2020-
96 p. : il. (algumas color.) ; 30 cm.

Orientador: Dr. André Luiz Peron Martins Lanna

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2020.

1. Palavra-chave01. 2. Palavra-chave02. I. Dr. André Luiz Peron Martins Lanna. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Utilização de operações de refatoração para aprimoramento de variabilidade em sistemas de software

CDU 02:141:005.6

Ronyell Henrique dos Santos

Utilização de operações de refatoração para aprimoramento de variabilidade em sistemas de software

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 28 de fevereiro de 2020:

Dr. André Luiz Peron Martins Lanna
Orientador

M.Sc. Ricardo Ajax Dias Kosloski
Convidado 1

M.Sc. Cristiane Soares Ramos
Convidado 2

Brasília, DF
2020

Resumo

A fase de manutenção é a fase mais onerosa no ciclo de vida de um software. A variabilidade tende a aumentar a manutenibilidade de um sistema, isto é, aumentar o percentual de correção de falhas ocorrerem com êxito. Formas de se aplicar variabilidade em um sistema vão desde o desenvolvimento de um código no paradigma orientado a objetos com a utilização de padrões de projeto ou de conceitos do paradigma (como por exemplo a herança), até o desenvolvimento de uma linha de produtos de software. Contudo, nem sempre um software é construido considerando sua fase de manutenção, logo, sua manutenibilidade tende a ser crítica nesse caso. A refatoração tenta mitigar alguns desses problemas pois ela de maneira controlada modificar o código para que haja uma melhoria em sua estrutura sem alterar o comportamento externamente observável. O presente trabalho tem o objetivo de aprimorar variabilidade por meio da aplicação de operações de refatoração em sistemas de software. Com base nisso, é proposta a construção de uma ferramenta que identifique pontos de variabilidade, verifique quais operações de refatoração são aplicáveis, escolha algumas delas e indique as operações que farão o código obter uma melhora no *design* do sistema com base em sua variabilidade. Para tal foram elencadas propriedades que conseguissem extrair um padrão de situações onde a variabilidade pudesse encontrada e foram verificadas novamente após a aplicação manual do aprimoramento. Com isso as métricas não apontavam para o contexto de refatoração, evidenciando a possibilidade da utilização dessa abordagem para apontamento de aprimoramento do *design* da variabilidade.

Palavras-chaves: Variabilidade, Refatoração, Otimização.

Abstract

Maintenance is the most costly phase in the software life cycle. The variability tends to increase the maintainability of a system such it increases the percentage of successfully errors corrections. There are different manners to apply variability in a system ranging from developing a code in the object-oriented paradigm, by using design patterns or properties like inheritance, up to developing a software product line. Usually the development of a software does not consider the maintenance phase even knowing the software maintenability tends to be critical. The refactoring attempts to mitigate some of these problems by improving the code's structure and preserving its external behavior. The present work aims to refine variability through the application of refactoring operations in software systems. Based on this, it is proposed to build a tool that identifies points of variability, check which refactoring operations are applicable, choose some of them and indicate the operations that will make the code improve the system design based on its variability. For that, properties were listed that could extract a pattern of situations where the variability could be found and were checked again after the manual application of the improvement. As a result, the metrics did not point to the refactoring context, showing the possibility of using this approach to improve the variability design.

Key-words: Variability, Refactoring, Optimization.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

SBSE	*Search-Based Software Engineering*
AST	*Abstract Syntax Tree*

Sumário

1 Introdução

1.1 Problema

Na Engenharia de Software a variabilidade é conceituada como a capacidade de alterar ou personalizar um sistema de modo que quanto maior for a *variabilidade* de um artefato mais fácil é realizar mudanças nesse artefato e, por consequência, mais manutenível é o sistema (GURP; BOSCH; SVAHNBERG, 2001). De acordo com (DHILLON, 2002) "manutenibilidade corresponde à probabilidade de um artefato com falha ser restaurado para uma condição de trabalho adequada".

A dificuldade de fazer alterações em sistemas é um problema recorrente durante o ciclo de vida de software (KERIEVSKY, 2004). O grande esforço necessário para compreensão do código, necessidade de mudanças no *design* e na arquitetura para correção ou evolução do sistema e a falta de documentação técnica são fatores que implicam em tal dificuldade. Além da alterações citadas anteriormente há ainda a complexidade em encontrar melhores formas ou, pelo menos, formas adequadas para realizar tais modificações no software. As alterações deveriam ser realizadas tendo em vista alguns atributos de qualidade do software, em especial, os atributos que refletem a qualidade do código desenvolvido (PIZKA; DEISSENBÖCK, 2007). Esses dois fatores devem sempre ser considerados em conjunto ao manter um software o que nem sempre é feito. Há, portanto, a necessidade de um estudo mais detalhado sobre como tais aspectos podem ser considerados em cojunto com destaque especial para a implmentação de variabilidade em software. Esse trabalho pretende utilizar um arcabouço de operações e técnicas para aprimorar variabilidades em um sistema em manutenção, de forma a utilizar um *design* mais robusto.

Portanto, dado o contexto acima o objeto de estudo deste trabalho é a otimização da variabilidade de um sistema de software tendo em vista a melhoria de seus atributos de qualidade, em especial, aqueles relacionados à manutenibilidade por meio de uma ferramenta. Contudo, para realização de tais evoluções é necessário definir um conjunto de transformações capazes de alterar a estrutura de um código-fonte.

1.2 Justificativa

De acordo com (GAMMA et al., 1995), é essencial levar em consideração a possibilidade de mudanças de um software ou mesmo ficará suscetível às reformulações. Por este motivo, a reutilização de software é utilizada para se obter uma maior variabilidade no sistema e, em consequência, diminuir a quantidade de tempo e esforço no desenvolvimento

de um software.

Um caso especial do aumento da manutenibilidade aumentada de um software dá-se em Linhas de Produto de Software (LPS). Uma LPS é definida de modo a explorar o quanto for possível os artefatos em comum de uma família de aplicações (CLEMENTS; NORTHROP, 2002) e, por consequência, a manutenção realizada em tais artefatos é compartilhada por todos os produtos instanciados a partir da LPS (SCHACH; TOMER, 2000). Além da manutenibilidade aumentada, a antecipação dos riscos relativos a modificações proporcionada pela maior variabilidade do sistema faz com que custos referentes a tempo, esforço e recursos sejam minimizados (GAMMA et al., 1995).

A Engenharia de Software Baseada em Buscas (do inglês *Search-Based Software Engineering* - SBSE) tenta resolver problemas de Engenharia de Software de maneira otimizada (HARMAN; JONES, 2001). Isso significa que, dado um problema de Engenharia de Software que possa ser caracterizado como um problema de otimização (maximização ou minimização), a SBSE *busca* uma solução adequada dentre todas as possíveis soluções. Para encontrar tal solução diferentes técnicas e algoritmos são utilizados a citar, dentre eles, os algoritmos baseados em buscas exaustivas e em heurísticas. As soluções encontradas ao longo da busca modificadas reiteradamente o que propicia um conjunto de soluções possíveis para um mesmo problema (HARMAN et al., 2010).

Portanto, dada a necessidade de manter e evoluir software de maneira sustentável (*i.e.* com foco no atendimento à atributos de qualidade de software), acredita-se que a manutenibilidade se beneficiará à medida em que a variabilidade e o reuso de software forem considerados. A manutenção e evolução podem ainda se beneficiar das técnicas de SBSE se tais atividades puderem ser caracterizadas como problemas de otimização. Espera-se que esse trabalho consiga empregar as técnicas de SBSE em atividades de manutenção e evolução de modo que o emprego de tais técnicas sugira aos desenvolvedores as melhores soluções com relação à variabilidade do software.

1.3 Objetivos

1.3.1 Objetivo Geral

O objetivo geral deste trabalho é, dado um conjunto de operações mapeadas, definir um conjunto de operações que aplicadas a um sistema específico aprimorará sua variabilidade.

1.3.2 Objetivos Específicos

Com o propósito de atingir o objetivo geral, foram estabelecidos os seguintes objetivos específicos:

- Identificar modelos de representação do sistema;
- Identificar métricas e propriedades que são relacionadas a variabilidade;
- Identificar pontos de variabilidade dentro do modelo de representação do sistema;
- Verificar viabilidade por meio do código fonte de algum sistema de software.

1.4 Metodologia

O presente trabalho será desenvolvido de maneira sequencial, de modo que as tarefas definidas serão executadas de maneira linear, podendo haver modificações em objetos decorrentes de atividades anteriores.

A fase de análise da necessidade é descrita na presente seção, a finalidade desta etapa é, sobretudo, a definição do escopo do trabalho, também são são definidos os objetivos do trabalho e revisão de literatura. Na revisão de literatura, são investigados aspectos referentes a variabilidade de sistemas de software, técnicas de otimização, operações de refatoração, técnicas de representação de um sistema e trabalhos correlatos. No planejamento da solução é feita com elaboração do cronograma de atividades.

Na fase de *design* é onde a solução foi projetada, portanto, são feitas algumas definições a respeito da execução e a modelagem arquitetural da solução. Na fase de construção a solução é desenvolvida.

Após a solução estar construída, será realizada a verificação de viabilidade. Nesta fase será escolhido um sistema para ser submetido a ferramenta para que a solução seja verificada. Posteriormente a submissão de um sistema a solução, os dados de saída deverão ser comparados com os dados iniciais.

Segundo (GIL, 2002) uma pesquisa pode ser classificado quanto aos objetivos em exploratória, descritiva ou explicativa. A pesquisa foco deste trabalho é exploratória. Uma vez que a pesquisa exploratória ocorre quando o objetivo é obter um maior conhecimento a respeito de um problema. Quanto a natureza da pesquisa se tem uma pesquisa de natureza aplicada, já que envolve a solução de um problema específico (MORESI et al., 2003).

1.5 Organização do Trabalho

Este trabalho está dividido em seis capítulos. São eles: introdução, referêncial teórico, metodologia, desenvolvimento, resultados e considerações finais. Os capítulos estão descritos logo abaixo.

No capítulo 1 - introdução - são apontados os objetivos, o problema a ser resolvido, justificativa e metodologia a ser utilizada no trabalho.

No capítulo 2 - variabilidade - São identificados conceitos e técnicas referentes à variabilidade, padrões de projeto, reuso e linha de produtos de software.

No capítulo 3 - refatoração - são identificados métodos e técnicas para a refatoração de software. Também são explicitadas as técnicas de otimização genéricas e aplicadas a engenharia de software.

No capítulo 4 - proposta - é identificada a abordagem metodologica utilizada para a construção do trabalho. O planejamento do trabalho, o processo a ser seguido, as técnicas aplicadas e tecnologias utilizadas se encontram nesta seção.

No capítulo 5 - resultados - é apresentada a execução do que foi planejado na seção de metodologia. Os resultados da tentativa de aumento de variabilidade são analisados e comparados com o estado inicial.

No capítulo 6 - considerações finais - são apresentadas as conclusões sobre o trabalho e são elencados possíveis trabalhos futuros.

2 Variabilidade

Para (GURP; BOSCH; SVAHNBERG, 2001), variabilidade é a capacidade de se alterar ou personalizar um sistema, (SVAHNBERG; GURP; BOSCH, 2005, pag. 706) incrementa o conceito e diz que variabilidade "é a capacidade de um sistema de software ou artefato de ser eficientemente ampliado, alterado, personalizado ou configurado para uso em um contexto particular".

A necessidade de variabilidade vem da necessidade de um sistema manutenível (GURP; BOSCH; SVAHNBERG, 2001). Reúso e flexibilidade são as principais frentes da variabilidade na implementação da orientação a objetos, *frameworks* e linha de produtos de software. O reúso é a capacidade de se aproveitar softwares pré-existentes na construção de novos softwares (KRUEGER, 1992). Um software flexível tem como propriedade característica a facilidade de modificação e se diferencia de um software genérico pelo fato do software genérico poder ser utilizado em diversas situações sem a necessidade de ser modificado (PARNAS, 1979).

2.1 Linha de Produtos de Software

Segundo (APEL et al., 2016), linha de produtos de software se baseia na indústria. Após a era artesanal a indústria surgiu com a produção em grande escala, para suprir as necessidades da massa. Ao contrário das individualidades, que eram consideradas no artesanato, a produção em grande escala tem como característica a padronização. Para suprir a ausência da individualidade surge a customização em massa, que nada mais é do que a apresentação de opções a escolha do cliente para construção de um produto mais adequado à suas necessidades.

> A abordagem da linha de produtos de software fornece uma forma de customização em massa através da construção de soluções individuais baseadas em um portfólio de componentes de software reutilizáveis. (APEL et al., 2016, pag. 8)

O uso de linhas de produtos de software em uma empresa apresenta algumas vantagens e algumas desvantagens. Quanto as desvantagens vale destacar o alto valor de investimento incial, complexidade de gerenciamento da variabilidade e configuração complexa. Em relação as vantagens tem-se *tailor-made*, redução de custos, melhoria de qualidade, *time to market*. *Tailor-made* faz referência a customização ou "feito sob medida", ou seja, SPL's viabilizam adaptações para incremento de individualidades. É inegável a existência de redução de custos, no entanto, existe a necessidade de um alto investimento

inicial. A melhoria da qualidade se dá pelo fato da produção ser em massa, padronização de componentes e sistematização de testes. *Time to market* traz referência ao tempo que é necessário para ser feito um produto, portanto, em SPL's o tempo de construção é bem rápido, já que o software padrão já está pronto e, caso seja necessário adicionar novas funcionalidade ou realizer algumas mudanças, as alterações tendem a ser mais simples (APEL et al., 2016).

2.1.1 Processo

O processo de desenvolvimento de uma linha de produtos é composta por dois processos: domain engineering e application engineering. Além disso, segundo (APEL et al., 2016), existe uma divisão espacial que delimita o problema e a solução.

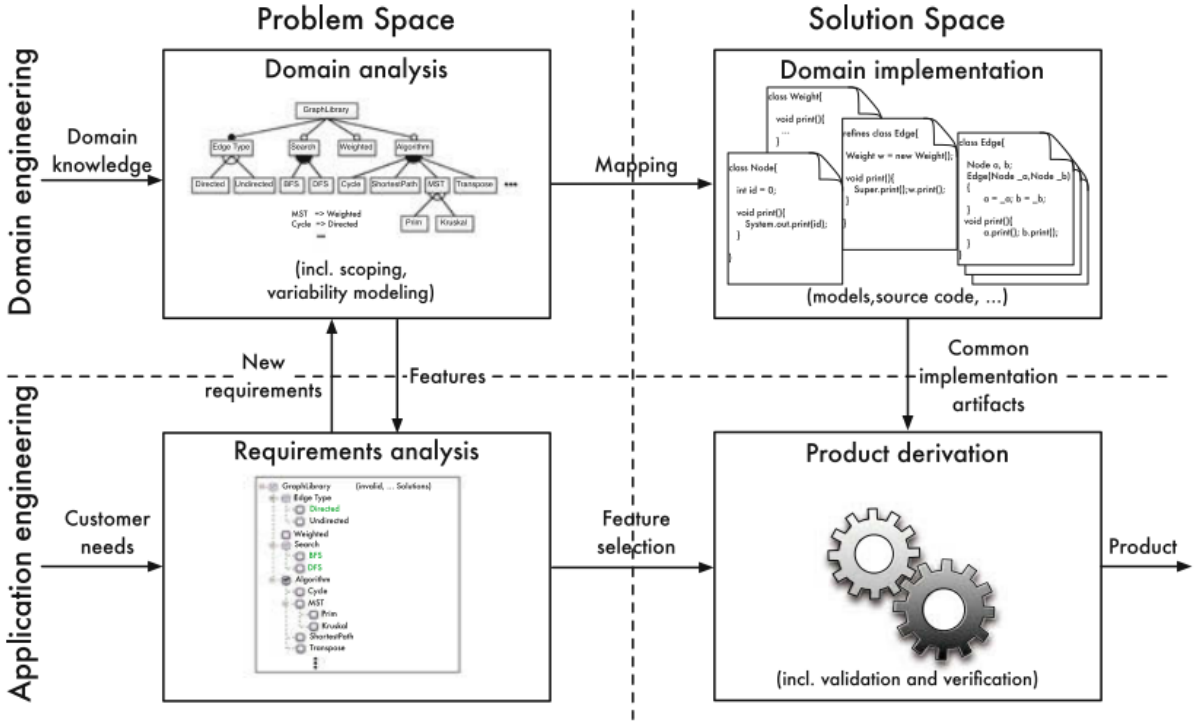


Figura 1 – Processo de uma linha de produtos. (Fonte: (APEL et al., 2016))

Na engenharia de domínio o objetivo é implementar artefatos que possam ser reutilizados, em outras palavras, é o desenvolvimento para reúso que é quando os artefatos são criados para serem reutilizados posteriormente. No processo de engenharia de aplicação é desenvolvido um produto específico. Este produto específico busca utilizar os artefatos gerados na engenharia de domínio, ou seja, é o desenvolvimento com reúso. (APEL et al., 2016)

No espaço do problema são especificados os requisitos dos domínios, tanto no processo referente a engenharia de domínio quanto na engenharia da aplicação. O espaço da solução envolve a contrução dos artefatos (APEL et al., 2016).

Na análise do domínio são deinidos os requisitos genéricos da SPL. A definição do escopo e a construção do modelo de features. Na análise de requisitos é feita a análise de requisitos de um produto, nessa etapa são mapeados os requisitos para *features* presentes na análise de domínio e algumas são selecionadas para a derivação do produto. Os requisitos podem contribuir para adição de novas *features* no modelo de *features* (APEL et al., 2016).

No implementação do domínio são construídos artefatos referentes aos requisitos da análise de domínio. O modelo de *features* é mapeado para estes artefatos, estes artefatos envolvem vários itens, como definições arquiteturais, código fonte e testes. A derivaçao de produto envolve a instanciação de um produto com base na seleção das *features*. A variante pode ser construída apenas com a composição dos artefatos construídos na implementação do domínio ou com acréscimo de novos artefatos para uma maior customização e aperfeiçoamento do produto (APEL et al., 2016).

2.1.2 Modelo de *Features*

Uma *feature* é um comportamento visível de um sistema de software. Elas orientam a estrutura, formas de reutilização e variação em um ciclo de vida de software. O modelo de *features* representa os relacionamentos das *features* de uma linha de produto e os relacionamentos este pode ser evidenciado por meio do diagrama de *features* que é uma notação para especificar as *features* e seus relacionamentos (APEL et al., 2016) .

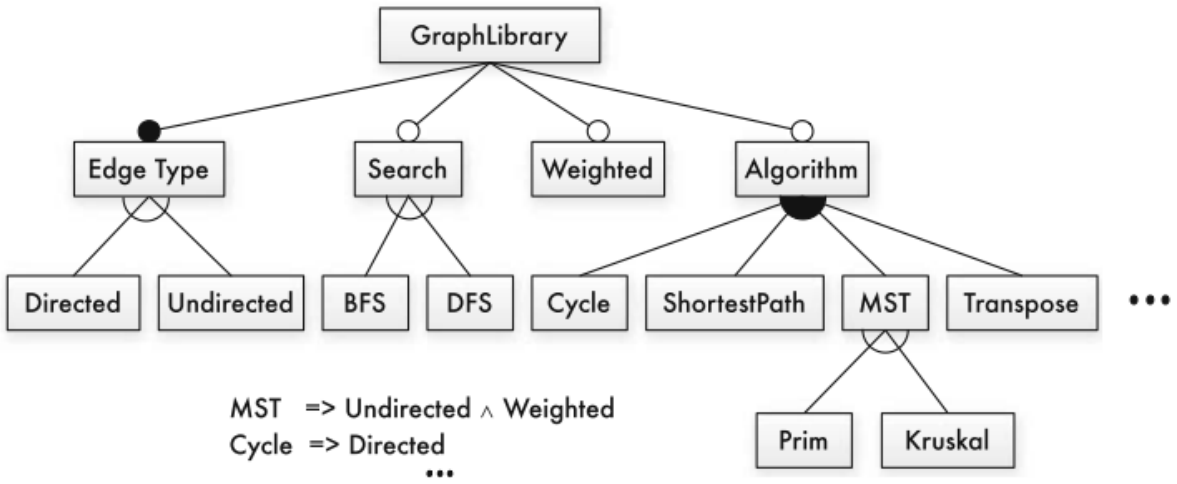


Figura 2 – Exemplo de diagrama de *features*. (Fonte: (APEL et al., 2016))

Os relacionamentos entre as *features* funcionam de forma hierarquica, onde as *features* filhas podem ser obrigatórias ou opicionais. Para que a *feature* filha possa ser selecionada a *feature* pai deverá ter sido selecionada previamente. Além disso, mais duas restrições são aceitas graficamente: escolha mutuamente exclusiva (operação XOR) e disjunção inclusiva (operação OR). Outras restrições referentes ao relacionamento podem

ser representados de forma proposicional. Todo diagrama de *features* tem uma forma proposicional equivalente (APEL et al., 2016).

2.1.3 Abordagens e Classificação

Segundo(APEL et al., 2016), SPL pode ser classificada em três dimensões, são elas: *binding time*, tecnologia utilizada e representação. Na dimensão *binding time* a variabilidade pode ser decidida em três momentos, o primeiro é em tempo de compilação, o segundo é em tempo de carregamento e o terceiro é em tempo de execução, o terceiro diferencia do segundo por conta da possibilidade de mudanças na configuração da variante em tempo de execução. Em tempo de compilação existe uma otimização do sistema, uma vez que não é necessário carregar todo o código da SPL, como é feito em tempo de execução e carregamento.

Na dimensão relacionada a tecnologia é categorizado qual tecnologia utilizada para derivação de produtos, são enumeradas duas possibilidades: abordagens baseadas em linguagem e abordagens baseadas em ferramenta. Nas abordagens baseada em linguagem a própria linguagem utiliza mecanismos para derivação de variantes. Nas abordagens baseadas em ferramenta são utilizadas ferramentas para gerenciar o processo de configuração de produtos (APEL et al., 2016).

A última dimensão refere-se a forma de construção das *features*. A representação pode ser realizada por anotação ou por composição. Em abordagens baseadas em anotação trechos de código são anotados como pertencentes de uma *feature*, a notação #ifnotdef é um exemplo de uso de abordagens anotativas, na derivação do produto todo o código que não pertence as *features* selecionadas são removidos. Em abordagens baseadas em composição as *features* são implementadas por meio de unidades compostas, ou simplesmente componentes, durante a derivação de produtos existe uma combinação para que o produto seja composto (APEL et al., 2016).

2.2 Padrões de Projeto

Segundo (APEL et al., 2016), existes variadas formas de se desenvolver código com variabilidade, estas técnicas são responsáveis por evitar utilização indiscriminada das estruturas condicionais por meio da parametrização, que muitas vezes se tornam confusas, e facilitar a rastreabilidade do código.

Padrões de projeto são definidos inicialmente (GAMMA et al., 1995), onde aborda soluções recomendadas para determinados contextos no desenvolvimento de software orientado por objetos. A orientação a objetos é um paradigma que utiliza a abstração de um contexto expecífico de algo do mundo real para implementar um sistema de software.

As técnicas mapeadas por (APEL et al., 2016) são parametrização, *design patterns*, *frameworks* e componentes e serviços. Apenas a segunda técnica será abordada com maior profundidade. Além disso, para fins de simplificação do problema somente alguns dos padrões serão descritos. Os padrões escolhidos foram: *strategy*, *template method*, *decorator* e *observer*.

2.2.1 *Strategy*

O *strategy* é um padrão de projeto, comportamental. Padrões comportamentais tem o propósito geral de atribuição de responsabilidade e algoritmos entre objetos. Este padrão tem o objetivo de definir e encapsular uma família de algoritmos e intercambiá-los. O padrão *startegy* pode ser aplicado em vários contextos. Um deles é a variação de algoritmos o outro é a substituição de estruturas condicionais em uma classe que define vários comportamentos. (GAMMA et al., 1995).

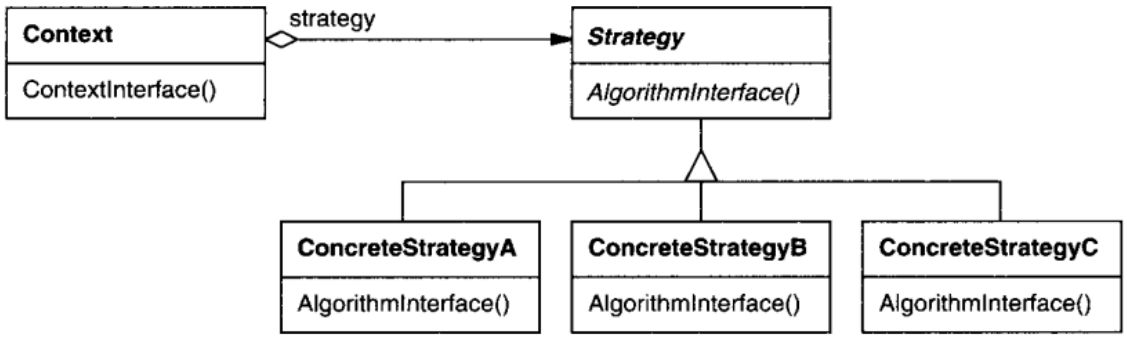


Figura 3 – Representação do Strategy (Fonte: (GAMMA et al., 1995))

2.2.2 *Template Method*

Assim como o *strategy*, o *template method* é um padrão de projeto comportamental. O objetivo de criar uma estrutura padrão de um algoritmo para as classes filhas, definindo o passo a passo, mas permitindo as subclasses alterar etapas (GAMMA et al., 1995).O *template method* é aplicável para implementar métodos invariantes, permitindo as classes filhas implementações específicas de trechos específicos.

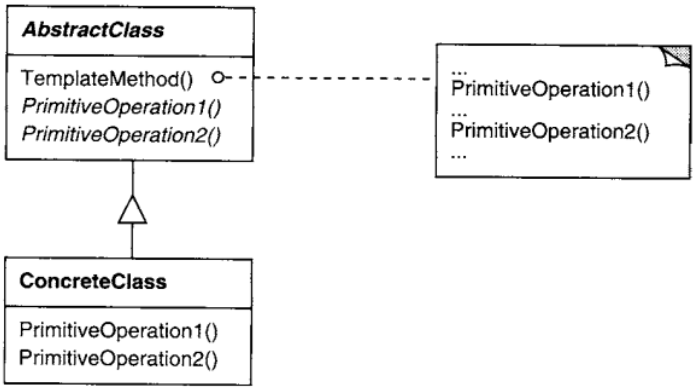


Figura 4 – Representação do Template Method (Fonte: (GAMMA et al., 1995))

2.2.3 *Decorator*

O *decorator* é um padrão estrutural e permite adicionar novos comportamentos a um objeto em tempo de execução, encapsulando objetos dentro de outros objetos. A aplicação deste padrão é indicada quando é necessário adicionar ou remover responsabilidades a um objeto em tempo de execução e para evitar a explosão de um grande número de subclasses (GAMMA et al., 1995).

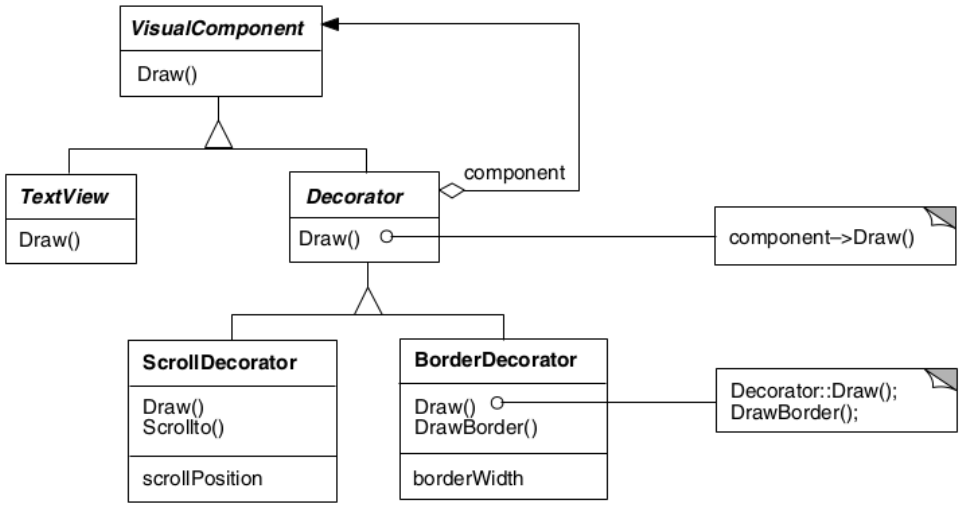


Figura 5 – Representação do Decorator (Fonte: (GAMMA et al., 1995))

2.2.4 *Observer*

O *observer* é um padrão comportamental que tem como objetivo notificar vários objetos por meio de uma assinatura, em outras palavras, o objeto observado mantêm uma lista de objetos que o observam e quando existe uma modificação no objeto passível de notificação aos objetos observadores, a notificação é enviada. A aplicabilidade desse método envolve situações que modificações em um objeto encadeia mudanças em outros

objeto. Outra aplicabilidade é na diminuição do acoplamento entre as classes em casos que se faz necessário um objeto notificar um outro (GAMMA et al., 1995).

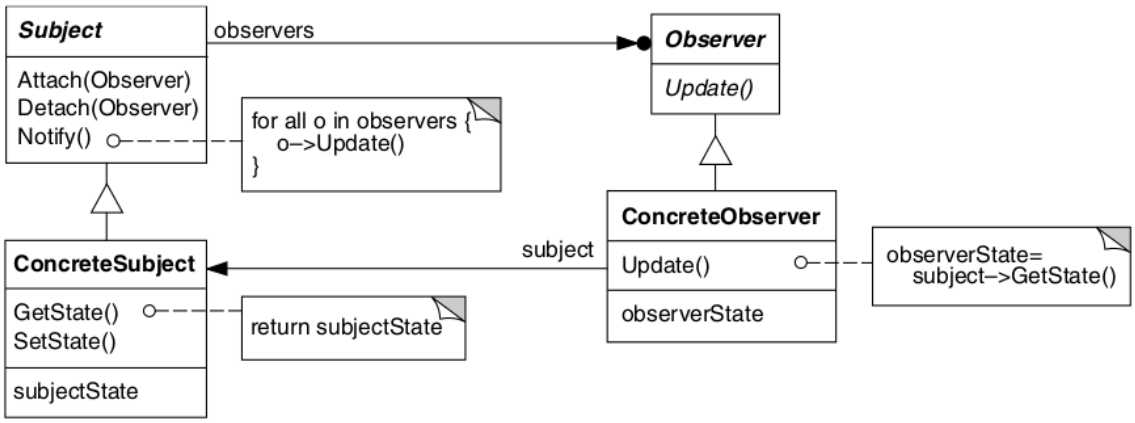


Figura 6 – Representação do Observer (Fonte: (GAMMA et al., 1995))

3 Refatoração

De acordo com (FOWLER, 1999), refatoração é o processo de modificação de software sem a que haja alterações de comportamento externo, promovendo uma melhoria da extrutura interna do código fonte. A necessidade de refatoração geralmente é identificada em várias ocasiões, algumas delas são quando há falta de entendimento do código pela estrutura não ser auto-explicativa, quando a complexidade de um trecho de código é alta e o entendimento é dificultado e quando há dificuldade de localização no projeto por questões estruturais.

3.1 Operações

Nas subseções seguintes as refatorações descritas são descritas da mais simples a complexas, onde as complexas envolvem a utilização de um conjunto de refatorações mais simples, as quatro últimas utilizam as refatorações mais simples em direcionadas a formação padrões de projeto.

3.1.1 Renomear método

Renomear método é utilizado para que o método comunique sua intenção (FOWLER, 1999). Geralmente, quando se torna necessário comentar sobre o nome do método essa refatoração deve ser utilizada . A sequência de passos de renomear método é simples:

1. Verificar se a assinatura do método é implementado em superclasses ou subclasses.
2. Declare novo método com o novo nome e modifique nos locais em que o antigo método era chamado.

3.1.2 Adicionar parâmetro

A necessidade de outros dados para executar as ações de um método é o motivo desta refatoração (KERIEVSKY, 2004). Para adicionar parâmetro são realizados os seguintes passos:

1. Verifique se a assinatura do método está presente em subclasses ou superclasses, se estiver presente os mesmos passos deverão ser realizados nelas;
2. Declare novo método com o novo parâmetro e copie o corpo do método antigo e compile;

3. Modifique o corpo do método antigo para que ele chame o novo e compile;
4. Procure as referências do método antigo e mude para o novo;
5. Remova o método antigo e compile.

3.1.3 Extrair parâmetro

Em oposição a receber parâmetros no construtor para atribuir valores a outro objeto, receber o objeto como parâmetro se faz mais coerente (FOWLER, 1999). Para isso são necessários os seguintes passos:

1. A atribuição de um atributo deve estar em um construtor. Caso não esteja, mova para um;
2. Aplique adicionar parâmetro.

3.1.4 Extrair método

Extrair método é uma operação de refatoração que tem como principais motivações-o tamanho grande de um método e a dificuldade de entendimento de um trecho de código (FOWLER, 1999). Esta operação consiste na extração de um fragmento de código de um método maior para que ele se torne auto-explicativo. Os passos para extrair método são:

1. Criar um novo método com nome auto-explicativo;
2. Copiar código a ser extraído para o novo método
3. Verificar variáveis locais e parâmetros;
4. Substitua o código por atribuição do novo método, caso necessário.

3.1.5 Substituir variável temporária por *query*

(FOWLER, 1999) afirma que variáveis locais tendem a se propagar e tem como consequência métodos mais longos. Essa operação tem como objetivo substituir uma variável que está guardando o resultado de uma expressão por um método que pode ser reutilizado em outros locais. A mecânica necessária para realização desta operação é:

1. Encontre variáveis locais atribuidas apenas uma vez;
2. Declare a variável temporária como final e compile;
3. Por fim, extraia a expressão e substitua pela assinatura do novo método.

3.1.6 Mover método

Dadas duas classes, A e B, onde B utiliza mais um método de A do que B é necessário realizar alguma refatoração para manter a coerência. A solução mais simples é utilizar a operação de mover método que, como diz o nome, move o método para a classe mais coerente (KERIEVSKY, 2004). Os passos necessários para a execução deste método são:

1. Verifique todos os elementos utilizados pelo método, se os elementos são utilizados apenas pelo método mova-os também.
2. Verifique se exite declarações na superclasse ou nas subclasses referente ao método, se exister não deve fazer a alteração;
3. Declare o método na classe destino;
4. Copie o código e faça ajustes para que funcione;
5. Determine a referência correta do objeto destino para o código movido;
6. Transforme o método origem em um método que realize a delegação;
7. Decida entre remover o método ou mantê-lo como um método de delação;
8. Caso remova, atualize as referências para o método movido.

3.1.7 Substituir condicional por polimorfismo

A existência de uma estrutura condicional que executa várias ações dependendo de um parâmetro é algo comum e pouco extensível. Uma alternativa a isso é substituir essa estrutura por polimorfismo (FOWLER, 1999). A sequência passos utilizadas para tal é a seguinte:

1. Extenda da classe onde está a estrutura condicional;
2. Para cada subclasse redefina o método e copie o código do ramo referente;
3. Remova o condicional e declare o método abstrato.

3.1.8 Subir método na hierarquia

A eliminação de duplicação de código é o objetivo fundamental desta operação (FOWLER, 1999). Logo, dado que duas subclasses de uma classe apresentam um método com o mesmo comportamento a operação de subir método na hierarquia espera manter apenas uma dessas estruturas na classe pai. A mecânica de refatoração tem os seguintes passos:

1. Verifica métodos idênticos
2. Se os métodos tem assinaturas diferentes, unifique as assinaturas.
3. Copie o corpo do método e cole na superclasse
4. Exclua os métodos da subclasse
5. Verifique onde o método está sendo chamado, caso necessário substitua para o novo nome do método.

3.1.9 Extrair interface

Um subconjunto de classes tem assinaturas similares e vários clientes utilizam esta parte da interface (FOWLER, 1999). Essa parte idêntica deve ser movida par uma interface própria. Os passos para realização desta refatoração são:

1. Crie uma nova interface;
2. Declare as operações comuns na interface;
3. Implemente a interface nas classes necessárias;
4. Ajuste o cliente para que o tipo da interface seja referenciado.

3.1.10 Substituir herança por delegação

Quando não é necessário a utilização de grande parte das funcionalidades providas pela superclasse a herança se torna mais um agravante do que um facilitador (FOWLER, 1999). Nesses casos, a delegação é mais adequada. Os passos requeridos para a relaização desta operação são:

1. Crie um atributo na subclasse com um atributo da superclasse;
2. Mude os métodos na sublasse para que usem os métodos delegados;
3. Remova a herança e crie um objeto da superclasse na subclasse;
4. Para cada método da antiga superclasse aplique a delegação.

3.1.11 Unificar interfaces

Quando existe a necessidade de processar objetos polimorficamente esta operação é utilizada. As assinaturas dos métodos públicos das subclasses, que não são herdados, são copiados para a superclasse e o comportamento é ajustado para nulo. A sequência de passos para realizar esta operação é:

1. Adicione a assinatura do método na superclasse;
2. Repita o passo 1 até que a assinatura da subclasse e da superclasse seja o mesmo.

3.1.12 Compor método

Um padrão simples é a composição de métodos, em que uma lógica robusta é substituida por um conjunto de métodos mais simples (KERIEVSKY, 2004). Existem alguns pontos negativos como a fragmentação do código e a dificuldade de depuração. Os passos para se realizar esta operação são:

1. Pense pequeno: métodos compostos geralmente tem de 5 a 10 linhas;
2. Remova duplicação e código morto
3. Comunique a intenção;
4. Simplifique;
5. Use o mesmo nível de detalhamento: quando extrair método faça com que todos tenham níveis de detalhamento similares.

3.1.13 Substituir Lógica Condicional por *Strategy*

De acordo com (KERIEVSKY, 2004) operações simples de refatoração como decompor condicional e decompor método podem sobrecarregar um classe com muitos métodos e por este motivo mover a variação para uma composição atrelado a uma herança pode ser uma solução. Este método tem a seguinte lógica: "Crie uma Strategy para cada variante e faça com que o método delegue o cálculo para uma instância de Strategy" (KERIEVSKY, 2004, p. 159).

A mecânica desse método de refatoração é composto por uma sequência de quatro passos. No entanto, é possível que no decorrer dos passos, o contexto do código, seja levado para um outro padrão. A sequência de passos para a substituir lógica condicional por *strategy* é:

1. Crie uma estratégia concreta;
2. Aplique mover método para a classe de estratégia e mantenha um método *delegate* na classe cliente, escolha entre passar contexto ou passar dados como parâmetro para a estratégia;
3. Crie no cliente uma instância da estratégia com a extração de parâmetros e atribua ao método *delegate*.

4. Aplique a técnica de substituir condicional por polimorfismo substituindo código de tipo por subclasses.

3.1.14 Formar *Template Method*

Duplicação de código é algo fácil de se encontrar em trechos de código fonte e muitas vezes esses trechos executam passos similares, porém, distintos. Logo, com uma sequência de passos invariáveis e com comportamento distinto. O formar *template method* tem a seguinte lógica "Generalize os métodos extraindo seus passos em métodos com assinaturas idênticas, então mova para a superclasse implementando o template method"(KERIEVSKY, 2004, p. 237)

Esta técnica é composta por uma sucessão de cinco passos, assim como no método anterior o contexto é variável e por este motivo a sequência de passos pode ser alterada resultando em algo diferente do padrão *template method*. Os passos para fomar *template method* são:

1. Encontre um método similar em uma hierarquia e aplique compor método
2. Aplique subir método na hierarquia para os métodos idênticos
3. Aplique renomear método em cada método único até que o corpo do método similar seja idêntico em cada subclasse.
4. Aplique renomear método no método similar para produzir assinatura idêntica em cada subclasse
5. Aplique subir método na hierarquia no método similar e defina métodos abstratos para cada método único.

3.1.15 Mover Embelezamento para *Decorator*

É comum adicionar novas responsabilidades ou novas funcionalidades em classes pré-existentes, o que aumenta a complexidade da classe com novos métodos, atributos. Geralmente, estes embelezamentos são utilizados apenas em alguns casos especiais. O decorator tenta solucionar este problema de forma elegante encapsulando a classe e adicionando um embelezamento (KERIEVSKY, 2004).

A mecânica dessa operação é composta por quatro passos, os passos para mover embelezamento para *decorator* são:

1. Identifique ou crie uma interface com o método de criar interface ou extrair interface.

2. Encontre a estrutura condicional que implementa o embelezamento e aplique substituir condicional por polimorfismo, se necessário aplique a operação formar *template method*.

3. Aplique substituir herança por delegação nas subclasses das classes embelezada.

4. Aplique extrair parâmetro e garanta que a lógica de atribuição da classe delegadora em seua tributo delegador.

3.1.16 Substituir Notificações *Hard-Coded* por *Observer*

Quando é necessário notificar várias instâncias ao mesmo tempo é possível utilizar o padrão *observer* sem a necessidade de notificações *hard-coded*. É importante destacar que este padrão diminui o acoplamento entre entidades muito acopladas ao mesmo tempo que pode deixar projetos mais complexos, caso exista um sequenciamento de notificações em cascata (KERIEVSKY, 2004).

A mecânica dessa operação contém seis passos. os passos para substituir notificações *hard-coded* por *observer* são:

1. Mova o comportamento de recepção das classes de notificadores com a operação mover método.

2. Aplique extrair interface na classe receptora criando uma classe observadora.

3. Todos os receptores devem se comunicar, exclusivamente, por meio dessa interface.

4. Escolha um notificador e aplica o método subir hierarquia criando a interface *subject*.

5. Atualize o observador para que ele possa se comunicar com o *subject* e então apague o notificador

6. Refatore o *subject* para que ele comporte uma lista de observadores e atualize o *subject* para que ele notifique uma lista de observadores.

3.1.17 Resumo

Com base nas operações de refatoração para padrões de projeto e nas operações mais simples de refatoração, é possível especificar as operações mais complexas por meio de uma sequência das operações mais atômicas. A Tabela 1 representa tal especificação.

3.2 Otimização

É comum que várias operações possam ser aplicadas em um mesmo contexto. Portanto, esta seção tem o objetivo de explicar alternativas para escolha das operações

Padrão	**Operação**	**Operações**
Strategy	Substituir lógica condicional por strategy	Mover método, extrair parâmetro, substituir condicional por polimorfismo
Template Method	Formar template method	Compor método, subir método na hierarquia, renomear método
Decorator	Mover embelezamento para decorator	Unificar interfaces, extrair interface, substituir condicional por polimorfismo, formar template method, substituir herança por delegação
Observer	Substituir notificações hard-coded por observer	Mover método, extrair interface, subir método na hierarquia

Tabela 1 – Resumo das operações utilizadas para refatoração para os padrões

de refatoração e critérios de escolha. Considerações referentes a otimização da escolha são definidos na seção algoritmos ambiciosos, na seção de programação dinâmica e na seção referente a *search-based software engineering*.

3.2.1 Algoritmos Ambiciosos

Segundo (KLEINBERG; TARDOS, 2006), algoritmos ambiciosos tem como objetivo otimizar algum tipo de critério em pequenos passos escolhe partes da solução sem considerar partes que serão agregadas a solução em passos futuros. É importante destacar que geralmente tais algoritmos produzem soluções próximas do ideal e que a dificuldade em manipular tais algoritmos é de provar o seu funcionamento.

3.2.1.1 Menor caminho

O objetivo dos *shortest paths*, ou menor caminho, é encontrar o caminho mais curto de um nó a outro nó em um grafo. Pesos são atribuidos às arestas e representam custos para caminhar de um nó a outro, estes custos são computados para cálculo deste caminho (KLEINBERG; TARDOS, 2006).

Dijkstra criou um algoritmo de nome homônimo que busca encontrar o caminho mínimo entre um nó i e um nó f. Para isso ele determina o caminho mais curto entre o nó i e todos os outros no grafo, tendo isso, é simples encontrar o custo e o caminho entre o nó i e o nó f (KLEINBERG; TARDOS, 2006).

Exemplo: um cliente u dejesa enviar informações para um servidor z via http. Os vértices são os roteadores por onde serão passados os pacotes e os pesos são representados pela latência. A Figura 7 corresponde a este relacionamento. Como o nó inicial é u e o algoritmo de Dijkstra tem como principal regra visitar os nós com menor peso, partindo

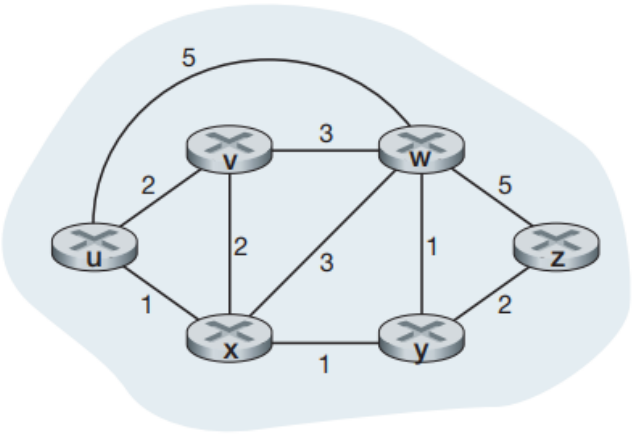


Figura 7 – Problema de roteamento (Fonte: (KUROSE; ROSS, 2006, pag. 270))

da origem, no primeiro passo obtemos menor caminho do nó u para todos os nós, iniciando pelo próprio u, onde é obtido a distância para ele mesmo (0) e para os nós que u não alcança (∞). Os passos subsequêntes utilizam a mesma ideia, sempre visitando os nós pela solução com menor peso no momento, formando uma árvore.

3.2.2 Programação Dinâmica

A programação dinâmica busca solucionar problemas de otimização por meio de análise combinatória. Este tipo de algoritmo, diferente dos algoritmos ambiciosos, utiliza a decomposição do problema, dividindo o problema em subsoluções, para decidir a melhor solução explorando todas as soluções possíveis (KLEINBERG; TARDOS, 2006).

Exemplo: um explorador sairá para escalar o Monte Everest, ele tem uma mochila que tem restrição de no máximo 6kg e deseja levar suprimentos que tenham o maior valor calórico possível, podendo levar apenas uma unidade de cada, existem quatro suprimentos *I={A, B, C, D}*, os pesos são *P={2, 2, 3, 3}* e o valor calórico *V={2, 3, 2, 4}*. A Tabela 2 demonstra como é a aplicação da programação dinâmica para resolução do problema da mochila. E a solução deste problema implica na escolha dos itens B e D, onde os pesos ficam abaixo do limite e maximiza o valor calórico.

	0	**1**	**2**	**3**	**4**	**5**	**6**
0	0	0	0	0	0	0	0
A	0	0	2	2	2	2	2
B	0	0	3	3	5	5	5
C	0	0	3	3	5	5	5
D	0	0	3	4	5	7	7

Tabela 2 – Programação dinâmica aplicada ao problema da mochila

3.2.3 *Search-Based Software Engineering*

Search-based software engineering é um campo da engenharia de software que busca aplicar otimização a problemas referentes a própria engenharia. Ela foi introduzida por (HARMAN; JONES, 2001) quando escreveu um trabalho auto-descrito como manifesto SBSE, onde descreve algumas aplicações, algoritmos utilizados e muitos dos desafios a serem resolvidos.

3.2.3.1 Abordagem

De forma geral, a construção de um algoritmo de busca pode ser construída por meio de dois componentes distintos: a representação e a função *fitness*. A representação necessita de dos dados referentes ao problema. A função *fitness* exige propriedades ou métricas como parâmetro (HARMAN et al., 2010). A Figura 8 mostra a configuração geral de uma abordagem SBSE.

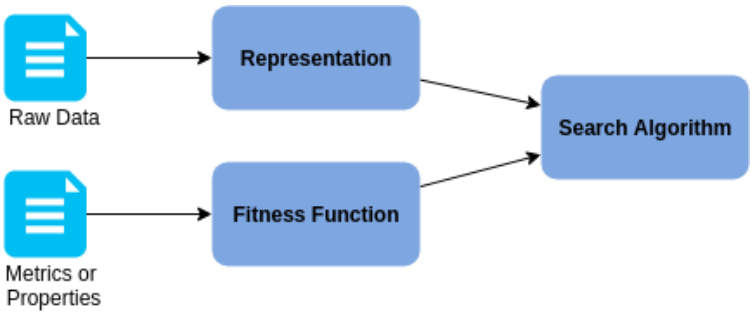


Figura 8 – Configuração de uma abordagem SBSE (Fonte: (HARMAN et al., 2010))

3.2.3.2 Algoritmos

Segundo (HARMAN et al., 2010), alguns algoritmos são utilizados comumente em SBSE, são eles: *Hill Climbing*, *Simulated Annealing* e *Genetic Algorithms*.

Hill Climbing, ou subida de encosta, é uma técnica de otimização que busca máximos locais. O algoritmo inicia com a escolha aleatória de uma solução, esta solução é comparada com os vizinhos, ambos se assemelham mas diferem em algum aspecto, se algum vizinho obter melhor adequação, de acordo com a função *fitness*, o ponto atual será, então, o ponto escolhido. Isso é feito em cada iteração, até que o número máximo de iterações estiver sido alcançado ou o objetivo tenha sido atingido. Existe a possibilidade de reinicialização, uma vez que as soluções tem caráter aleatório (HARMAN et al., 2010).

Assim como *Hill Climbing*, *Simulated Annealing*, ou recozimento simulado, é um algoritmo de otimização de busca local. No entanto, se baseia no recozimento físico onde é explicado que quando um material é submetido a ponto de fuzão e resfriado novamente, o material retorna ao estado sólido, contudo, as propriedades estruturais variam de acordo

com o resfriamento. Logo, tem como fator de variação a temperatura. O algoritmo se começa com uma solução inicial aleatória a temperatura é elevada e movimentação livre, a temperatura diminui aos poucos e quando congela este algoritmo se comporta como o anterior (HARMAN et al., 2010).

3.2.3.2.1 Algoritmos Genéticos

Genetic Algorithms, ou algoritmos genéticos, são algoritmos que buscam máximos globais, por este motivo apresentam soluções mais robustas e implementação mais complexa. Os algoritmos genéticos tem como base a teoria da evolução de Darwin onde a população atual é representada pelas soluções possíveis, gerações as populações sucessivas, onde soluções são representadas por cromossomos. Além disso, existem operadores como seleção, *crossover* e mutação (HARMAN et al., 2010). A Figura 9 demonstra o funcionamento genérico de um algoritmo genético.

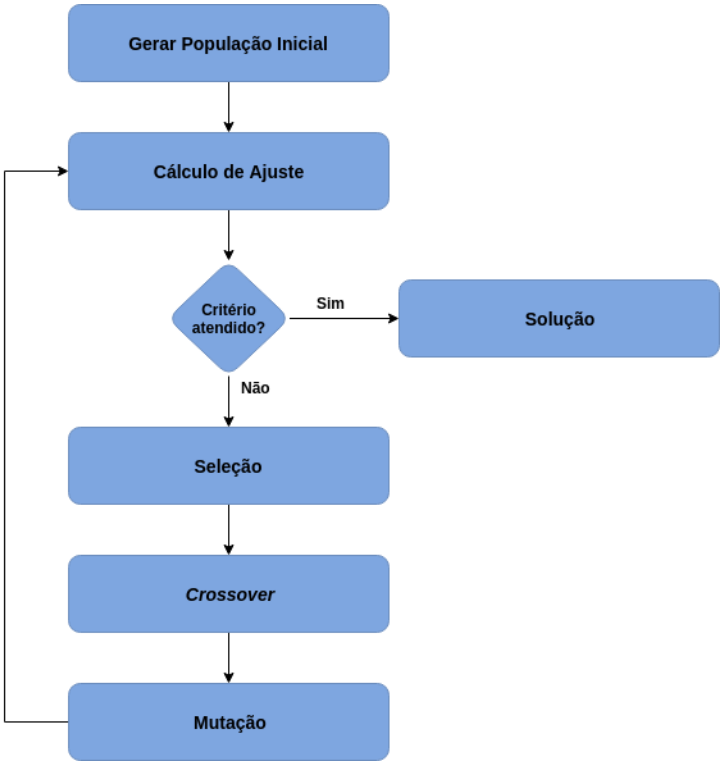


Figura 9 – Esquemático de funcionamento genérico de algoritmos genéticos

Inicialmente são gerados indivíduos, ou soluções, aleatoriamente. As soluções geralmente são representadas de forma binária. Depois, cada indivíduo é avaliado por uma função de ajuste definida. Se os critérios forem atendidos ou o tempo máximo for ultrapassado a solução ou um conjunto de soluções são selecionadas. Caso os critérios não sejam atendidos a população atual é submetida a alguns operadores, como seleção, *crossover* e mutação (HARMAN et al., 2010).

Na seleção é, geralmente, utilizado torneios *n-way*, onde é escolhida uma amostragem aleatória com n elementos, a solução individual mais adequada é escolhida e esse processo é repetido algumas vezes para que seja feita a reprodução. No *crossover* são escolhidos dois indivíduos i1 e i2, após isso, é escolhido um número k, que representa o tamanho do corte realizado nos indivíduos, a resultante deste corte é permutada entre os indivíduos i1 e i2 gerando dois novos indivíduos c1 e c2. A mutação, assim como na biologia, ocorre de forma aleatória e é responsável por introduzir novas propriedades na população e garante sua diversidade, este operador é empregado com o auxílio de uma taxa de mutação relativamente pequena (HARMAN et al., 2010).

4 Proposta

Segundo (SOMMERVILLE, 2011), a manutenção de um software consome entre 50% e 65% dos recursos financeiros direcionados a um sistema. A variabilidade é um fator importante em produtos de software, uma vez que a variabilidade tende a aumentar a manutenibilidade e em consequência a isto o custo é diminuido (DHILLON, 2002).

A proposta do presente trabalho envolve a construção de uma ferramenta para extração de variabilidade em sistemas de software. A ferramenta desenvolvida tem como entrada código fonte de um sistema desenvolvido. Após isso, são geradas suas representações para que posteriormente possam ser analisadas para determinação e aplicação das operações de refatoração. No passo seguinte o sistema obtém uma nova configuração representativa e, consequentemente, um código fonte diferente, potencializando sua variabilidade. A Figura 10 representa o fluxo da solução proposta.

O objetivo deste trabalho é extrair variabilidade de um sistema de software por meio da utilização de um arcabouço de operações de refatoração.

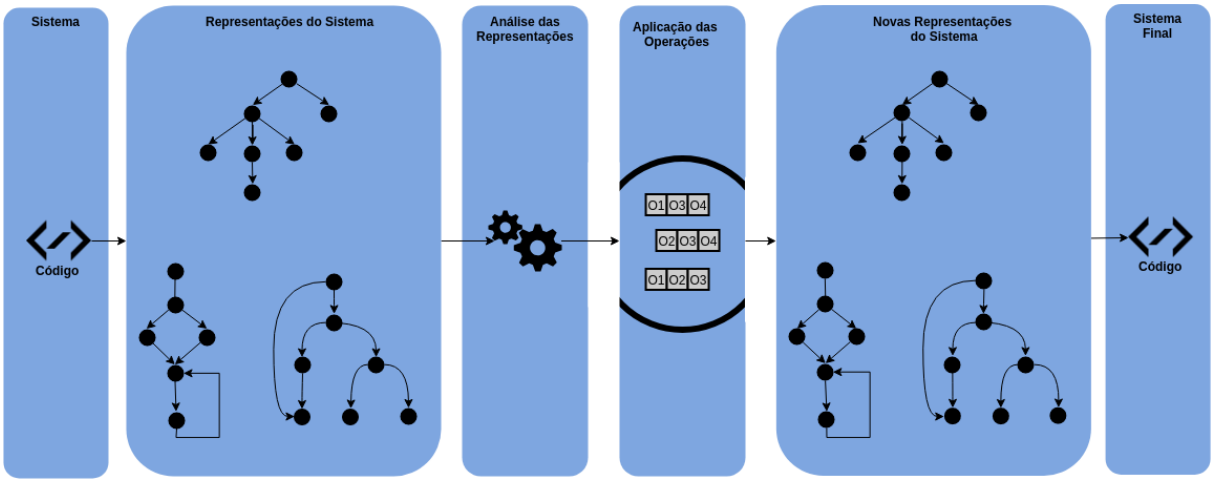


Figura 10 – Proposta de solução (Fonte: Autor)

4.1 Representação do sistema

Segundo (FAIRLEY, 1978), a análise estática de código busca encontrar informações sobre a estrutura do código fonte de um software. A análise estática é uma análise feita sem necessidade da execução do programa. As informações obtidas neste tipo de análise vão desde mensagens de erros de sintaxe até o mapeamento de sub-rotinas chamadas por uma rotina.

De acordo com (BRUMLIK; VANNIN, 1970), uma ferramenta de análise de código fonte realiza uma sequência de procedimentos. Em geral, o primeiro passo é a identificação do código fonte a ser analisado que se dá por meio de reconhecimento dos arquivos a serem examinados.

Em seguida é preciso criar um modelo do sistema que é um conjunto de representações abstratas do software ou simplesmente representações intermediárias do sistema como *Abstract Syntax Tree* (AST), *flow graph* e *call graph*. Ambos utilizam grafos como base para suas representações. Um grafo consiste em um conjunto de vértices e um conjunto de arestas, onde o propósito das arestas é interligar dois vértices (KLEINBERG; TARDOS, 2006). A AST é uma representação estrutural do código fonte, muito utilizada em compiladores.

No *call graph* os vértices representam procedimento, no caso de aplicações orientada a objetos os nós são os métodos, e as chamadas de outros procedimentos são representadas por setas. Em um *flow graph* os vértices representam as instruções feitas e as setas representam os caminhos que podem ser executados em uma aplicação (FAIRLEY, 1978).

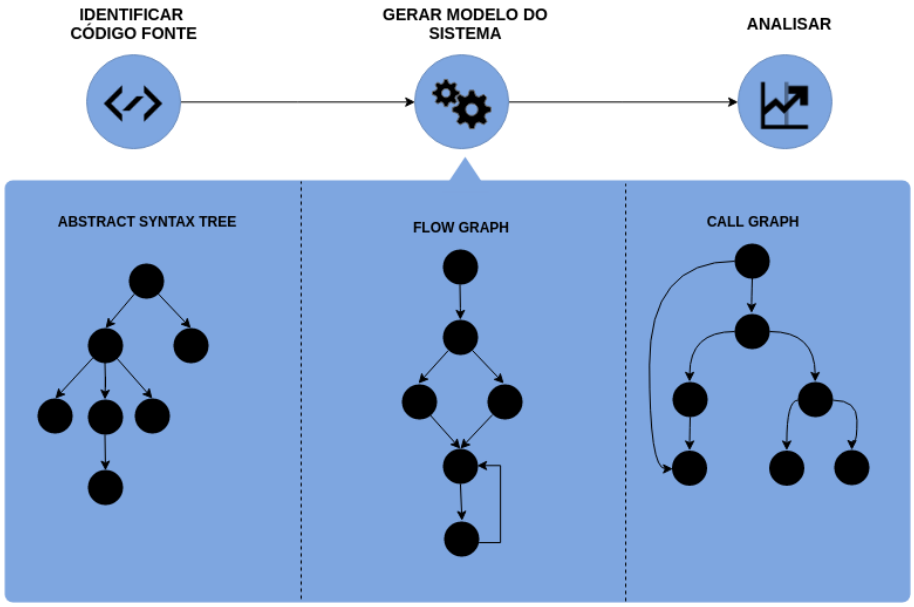


Figura 11 – Representação do processo de análise estática. Baseado em (RFLEMING, 2018).

A última etapa a ser realizada é a análise do sistema, seja em busca de alguma anomalia, verificação de padrões de codificação ou outros objetivos específicos. A Fig. 11 é uma demonstração do processo de análise estática e suas representações intermediárias. A representação do sistema é um passo fundamental para o representação do problema, dado isso, o sistema será representado com o apoio das representações intermediárias do processo de análise estática de código fonte.

4.2 Métricas relevantes

As métricas escolhidas para análise do processamento foram estabelecidas com base em experimentos empíricos e com base na descrição das operações e dos padrões de projeto apresentados nas seções anteriores. Algumas métricas, na verdade, são características do código, como a coesão que é uma característica do código e que tem diversas métricas para seu cálculo.

Dado o catálogo de métricas A, foram escolhidas seis métricas para utilizar como base inicial da pesquisa por fatores referentes ao contexto observado nas situações aplicavéis nas operações de refatoração. São elas

- SLOC: Substituir lógica condicional por *strategy*, Formar *template method*, Mover embelezamento para *decorator*
- CD: Formar *template method*, Mover embelezamento para *decorator*, Mover embelezamento para *decorator*
- CC: Substituir lógica condicional por *strategy*, Formar *template method*
- NOM: Substituir lógica condicional por *strategy*, Formar *template method*, Mover embelezamento para *decorator*
- LCOM: Substituir lógica condicional por *strategy*
- CBO: Formar *template method*, Substituir notificações *hard-coded por observer*

4.3 Arquitetura da Solução

A ferramenta é constituída por alguns módulos. No total são seis módulos com atribuições específicas para o cumprimento da solução proposta. A Figura 12 especifica todos os módulos presentes na ferramenta e seus relacionamentos.

O módulo de manipulação de entrada é responsável por obter o código fonte a ser submetido pelo restante do processamento. Recebe parâmetros via terminal que determinam o local de carregamento do código fonte e o local de destino do código modificado.

A extração das representações tem como objetivo utilizar o código fonte que está em memória para construir as representações do sistema, as representações construídas neste módulo são limitadas a três: *AST*, *flow graph* e *call graph*. Este módulo também tem a responsabilidade de inferir tipos às variáveis e retornos de métodos.

A análise do sistema utiliza como parâmetro tanto as representações produzidas pelo módulo de extração quanto o código fonte para que se possa inferir alguns *code smells* e, em consequência, algumas operações a serem aplicadas.

No módulo de aplicação das operações, os procedimentos de refatoração inferidos no módulo de análise serão executados. Geralmente, uma série de operações serão recomendadas no módulo de análise.

A manipulação de saída tem como objetivo salvar o código modificado. A estrutura do código de entrada será mantida, se possível, uma vez que as operações podem fazer com que algumas novas estruturas, como classes, sejam excluídas ou inseridas.

O módulo *core* é responsável por interligar todos os outros módulos, portanto, este módulo é um centralizador de mensagens, atuando como uma indireção para que os outros módulos se comuniquem sem uma interdependência entre eles, uma vez que a interface de comunicação é definida pelos parâmetros e retornos que o módulo disponibiliza.

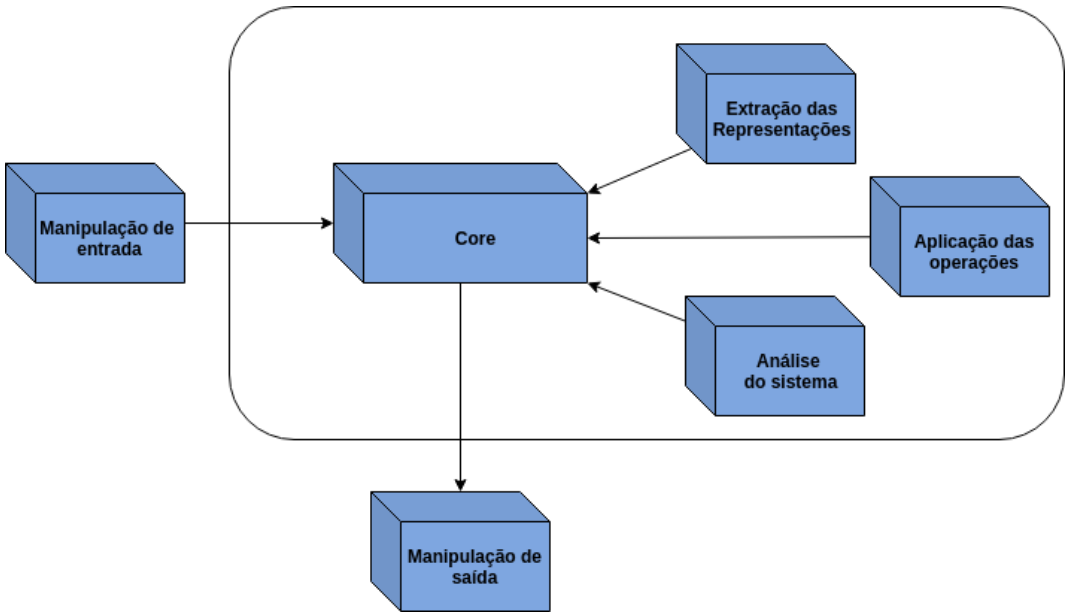


Figura 12 – Módulos da ferramenta (Fonte: Autor)

5 Resultados

Durante a primeira parte do desenvolvimento do trabalho, buscou-se a validação de algumas das ferramentas, técnicas e modelos investigados. Foram observados alguns padrões de comportamento, por meio das representações do sistema antes e depois da aplicação das operações de refatoração, e por fim, algumas limitações das ferramentas que serão utilizadas foram identificadas.

5.1 Padrões observados

5.1.1 Representações

No grafo de chamadas foram utilizados três elementos: *containers* que representam as classes do software observado, retângulos que simbolizam os métodos e as setas que representam os relacionamentos entre classes e métodos. Foi feita uma adaptação do grafo de chamadas para inclusão da representação de estruturas hierarquicas para extrair maiores informações a respeito da estrutura do código analisado.

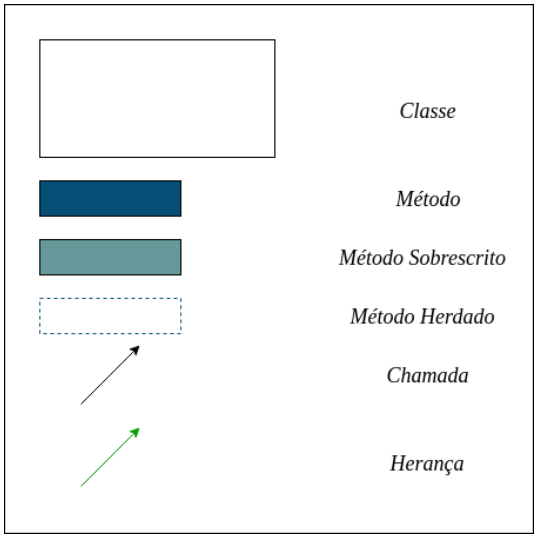


Figura 13 – Legenda do *call graph* (Fonte: Autor)

Os métodos são representados por três tipos de retângulos diferentes. O azul escuro indica que o método é declarado, inicialmente, naquela classe, o azul claro informa que o método é herdado e sobrescrito na classe em questão e por último o branco com borda pontilhada representa um método apenas herdado. Quanto as setas, a de cor verde representa herança entre as classes e de cor preta as chamadas entre os métodos.

No grafo de fluxo são utilizados quatro elementos: elipse que representa o início e o fim do método, losango representa uma estrutura condicional, retângulo que representa

as linhas do código fonte e as setas representando o fluxo da função.

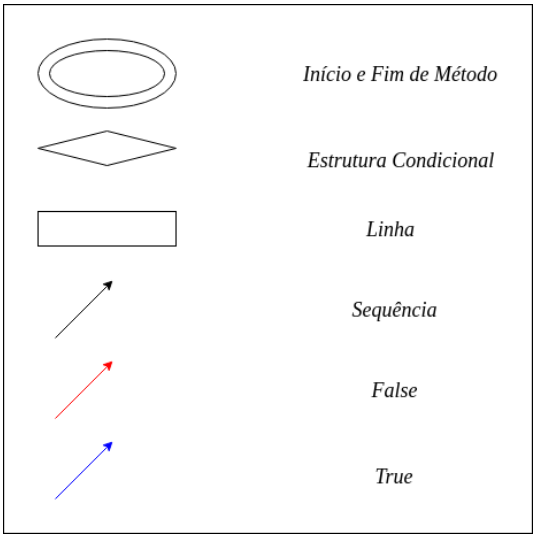


Figura 14 – Legenda do *flow graph* (Fonte: Autor)

As setas representam o fluxo do código entre as estruturas condicionais e as demais linhas do código. As linhas setas pretas representam o fluxo comum, sem decisões a serem realizadas, as setas vermelhas são utilizadas para informar que quando a condição da estrutura de decisão for falsa ela percorrerá aquele caminho e a seta de cor azul tem o mesmo princípio da flecha vermelha, no entanto, representa a condição verdadeira.

5.1.2 Análise das métricas

Foi observado o comportamento das métricas para cada uma das operações elicitadas e cada métrica foi analisada com base na relevância na situação apropriada para refatoração. As operações foram classificadas com grau de interferência muito alto, alto, indiferente, baixo e muito baixo. Ou seja, quanto mais alta uma métrica mais relevante ela é quando atinge números altos, em contrapartida uma métrica com um indíce baixo indica que o comportamento é inversamente proporcional ao comportamento. Métricas que são classificadas como indiferentes são as métricas que não se mostraram consideráveis para as situações observadas.

Ao mesmo tempo foi categorizado em qual escopo a operação se encontrava, isto é, em qual âmbito a operação seria analisada. Foram observados três possíveis escopos: método, classe e módulo. Se a operação se apresentar em um escopo de método as análises para determinar a necessiadade ou não da operação deve ser feita apenas em nível de método, sem considerar o todo. Isso é aplicável aos outros escopos, em caso de classe, os módulos devem ser desconsiderados para análise mas os métodos devem ser incluidos, uma vez que os métodos fazem parte da classe.

Com base nas análises, a Tabela 3 descrimina qual o comportamento das métricas em cada uma das situações onde as operações são aplicáveis.

	CC	**SLOC**	**NOM**	**CD**	**LCOM**	**AC**	**SC**
Renomear método							MT
Adicionar parâmetro							MT
Extrair parâmetro							MT
Extrair método	AL	MA		AL			MT
Substituir variável temporária por query		AL		MA			MT
Mover método					MA		CLS
Substituir condicional por polimorfismo	MA	AL					CLS
Subir método na hierarquia		MA				MA	MD
Extrair interface		MA				MA	MD
Substituir herançapor delegação		MA				MA	MD
Unificar interfaces		MA				MA	MD
Compor método	AL	MA					MT
Substituir lógica condicional por strategy	MA	AL	AL				MD
Formar templatemethod	AL	AL	AL	MA		AL	MD
Mover embelezamento para decorator	AL	AL	MA	MA			CLS
Substituir notificações hard-coded por observer						MA	MD

Tabela 3 – Interferência das métricas nas situações observadas

5.1.2.1 Renomear método

- Escopo: método
- Resultado: observa-se que na operação renomear método as métricas não diferem antes e depois da operação uma vez que esta operação está relacionada ao domínio que está sendo utilizada, portanto, é possível dizer que em sua totalidade essa operação de refatoração não está amplamente relacionada com as métricas elencadas.

5.1.2.2 Adicionar parâmetro

- Escopo: método

- Resultado: das métricas elencadas, nenhuma foi afetada de forma conclusiva por essa refatoração. Dessa forma, é possível declarar que todas tem um comportamento indiferente (ID) quanto a essa operação.

5.1.2.3 Extrair parâmetro

- Escopo: método
- Resultado: da mesma forma apresentada em *adicionar parâmetro*, das métricas elencadas, nenhuma métrica foi afetada de forma conclusiva por essa refatoração. Dessa forma, é possível declarar que todas tem um comportamento indiferente (ID) quanto a essa operação.

5.1.2.4 Extrair método

- Escopo: método
- Resultado: observa-se que a métrica com maior mudança é a *SLOC*, uma vez existe uma certa proporcionalidade entre a quantidade de linhas e as resposabilidade de um método. Logo, *SLOC* tem relevância muito grande para determinação da execução dessa operação e por este motivo *SLOC* tem um grau de interferência muito alto (MA). Outras métricas que tem um nível formidável são: *CC* e *CD*, ambos com interferência alta (AL).

5.1.2.5 Substituir variável temporária por *query*

- Escopo: método
- Resultado: a duplicação de código *CD* tem uma relevância muito alta (MA), uma vez que a atribuição da variável temporária pode ser repetida diversas vezes. Além disso, variáveis temporárias tendem a deixar os métodos maiores e a influência da métrica *SLOC* é alta (AL).

5.1.2.6 Mover método

- Escopo: método
- Resultado: o acoplamento tende a ser a característica que mais afeta a escolha dessa operação. No entanto, não foi possível a aferição de métricas relacionadas ao acoplamento. Além disso, a coesão (LCOM) também tem papel relevante nessa operação.

5.1.2.7 Substituir condicional por polimorfismo

- Escopo: método
- Resultado: a complexidade ciclomática CC tende a ser muito alta (MA) nesses contextos, resultando em vários ramos, por conseguinte às estruturas condicionais um alto (AL) valor de $SLOC$ é comum.

5.1.2.8 Subir método na hierarquia

- Escopo: módulo
- Resultado: a duplicação de código CD e o acoplamento entre as classes envolvidas tendem a ser altas em um contexto onde essa operação é indicada.

5.1.2.9 Extrair interface

- Escopo: módulo
- Resultado: a duplicação de código CD e o acoplamento entre as classes envolvidas tendem a ser altas em um contexto onde essa operação é indicada.

5.1.2.10 Substituir herança por delegação

- Escopo: módulo
- Resultado: a duplicação de código CD e o acoplamento entre as classes envolvidas tendem a ser altas em um contexto onde essa operação é indicada.

5.1.2.11 Unificar interfaces

- Escopo: módulo
- Resultado: a duplicação de código CD e o acoplamento entre as classes envolvidas tendem a ser altas em um contexto onde essa operação é indicada.

5.1.2.12 Compor método

- Escopo: método
- Resultado: uma lógica robusta comumente envolve uma muito alta $SLOC$, podendo haver uma alta (AL) quantidade de lógicas condicionais tedem a existir.

5.1.2.13 Substituir lógica condicional por *strategy*

- Escopo: método
- Resultado: a estrutura condicional é uma forma de gerar ramificações em um código, gerando uma maior complexidade ciclomática. Isto posto, é evidente que a métrica *CC* é de grande relevância para esta operação. Portanto, a interferência da métrica *CC* é muito alta (MA) para a execução dessa operação. As métricas *SLOC* e *NOM* podem ter uma relevância alta (AL) em alguns casos.

5.1.2.14 Formar *template method*

- Escopo: módulo
- Resultado: um dos benefícios gerados por essa refatoração é a remoção de código duplicado, logo, a relevância da métrica *CD* é muito alta para esta operação. As métricas *NOM*, *SLOC* e *CC* tendem a ter uma importância considerável (AL) para escolha dessa operação de refatoração. Além do acoplamento *CBO*, uma vez que envolve uma estrutura hierarquica inicialmente.

5.1.2.15 Mover embelezamento para *decorator*

- Escopo: classe
- Resultado: duplicação de código é removido *CD* dos embelezamentos presentes, ou seja, a duplicação de código é um fator de muito alta relevância (MA), outro fator de muita relevância é a quantidade de métodos *NOM* da classe. Grande quantidade de estruturas condicionais *CC* e *SLOC* e grande fator de coesão *LCOM* também tem alta influência na escola dessa operação (AL).

5.1.2.16 Substituir Notificações *hard-coded* por *observer*

- Escopo: módulo
- Resultado: o acoplamento *CBO* entre as classes envolvidas tendem a ser muito altas em um contexto onde essa operação é indicada.

5.2 Funcionamento da ferramenta

Com base nos padrões observados nas métricas e nas representações foi criado, então, uma ferramenta que possa indicar operações possíveis para algumas situações encontradas em código.

5.2.1 Entradas

A ferramenta é executada para analisar um software que se encontra em um diretório. Portanto, a entrada da ferramenta é basicamente um diretório. Tal caminho pode ser relativo, quando se refere a partir do diretório atual, ou caminho absoluto, que se inicia na raiz do sistema operacional.

5.2.2 Visualizações

As visualizações foram utilizadas para exibir gráficamente os resultados das medições e das representações, deixando mais claro a situação do sistema. Um exemplo pode ser visto com as métricas SLOC, LCOM, NOM e CC na Figura 15a e na Figura 15b. Portanto, as visualizações são saídas intermediarias antes da saída final.

(a) Gráfico de métricas

(b) Gráfico de duplicação

Figura 15 – Visualizações (Fonte: Autor)

5.2.3 Saídas

Como as métricas escolhidas inicialmente não são suficientemente exclusivas de cada um dos contextos para indicação de uma única operação ser realizada. Não é possível indicar apenas uma operação, portanto, a saída consiste em uma listagem de classes e métodos e a porcentagem de probabilidade acerto, isto é, dado os valores aferidos, é mostrado ao usuário a porcentagem da operação ser aplicavel naquele trecho do sistema.

A Figura 16 apresenta a tabela referente a uma operação, onde método *Account.overdraft_change* apresenta 100% de correspondência com o contexto para aplicação da operação substituir variável temporária por query.

replace temp variable with query

	item	percent	match
0	AccountType.__init__	0.0	[]
1	AccountType.overdraft_change	0.5	[length]
2	AccountType.is_premium	0.0	[]
3	Account.__init__	0.0	[]
4	Account.overdraft_change	1.0	[length, cyclomatic_complexity]
5	Account.bank_change	0.0	[]

Figura 16 – Tabela de resultadosx (Fonte: Autor)

5.2.4 Experimento

Com base na adaptação de um exemplo feito por (KERIEVSKY, 2004, pag. 163-173), foi realizado um experimento para extração das representações do sistema. O exemplo busca aplicara operação de refatoração substituir lógica condicional por *strategy*.

5.2.4.0.1 Representações

Inicialmente, existe apenas uma classe (*Loan*) que detem um construtor e um método de cálculo de capital. No entanto, a existência de um método (*capital*) com uma estrutura condicional muito complexa fez com que a aplicação de refatoração fosse proposta. A Figura 17 representa as chamadas a funções, no caso nenhuma chamada é feita, e a Figura 18 representa o fluxo do método *capital* que parece ser horizontal, e que as estruturas condicionais são responsáveis por esta estrutura.

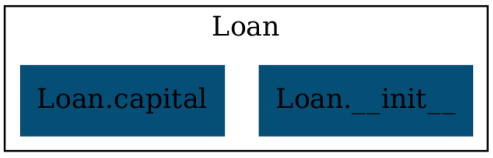


Figura 17 – *Call Graph* inicial

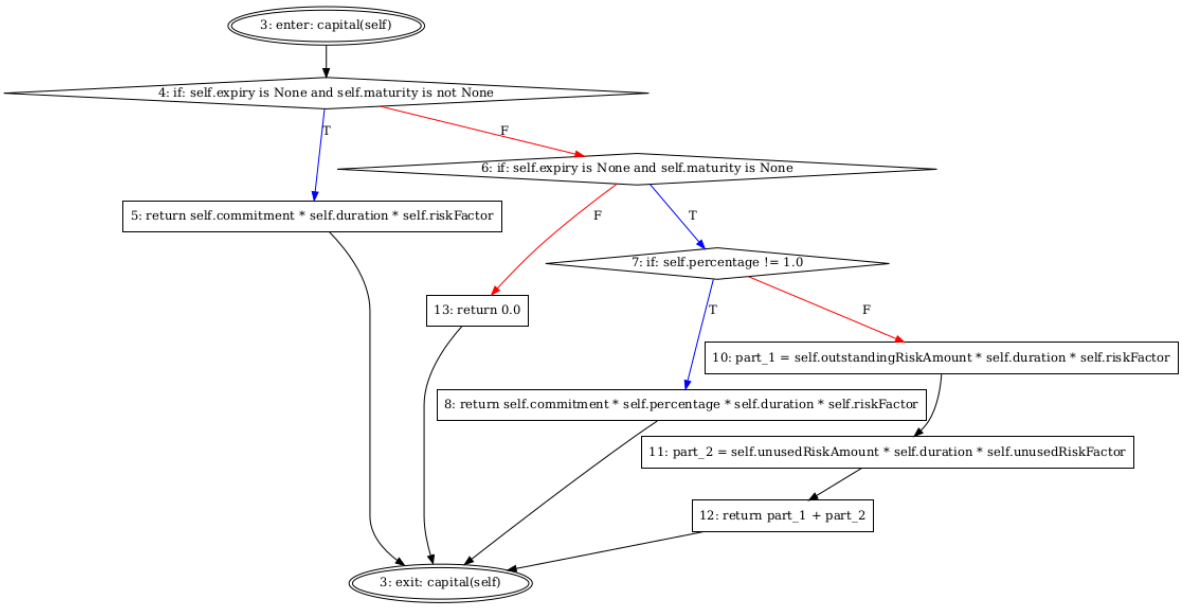


Figura 18 – *Flow Graph* inicial

A Figura 19 representa as chamadas método a método. É observado um aumento grande no número de classes e dos relacionamentos entre elas. o método *capital* na classe *Loan* funciona apenas como uma indireção para a estratégia de cálculo utilizada. A Figura 20 mostra todos os métodos derivados do método *capital*, é observado um padrão pouco horizontal da estrutura de forma gráfica sendo preferível uma verticalidade dos métodos.

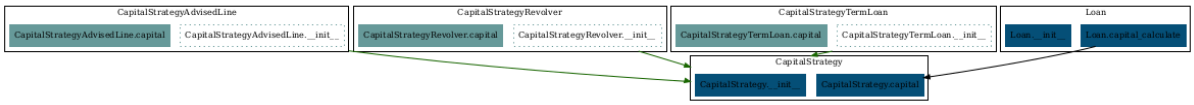
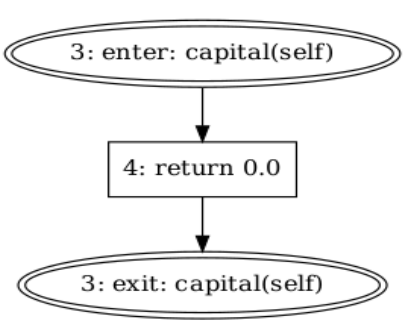
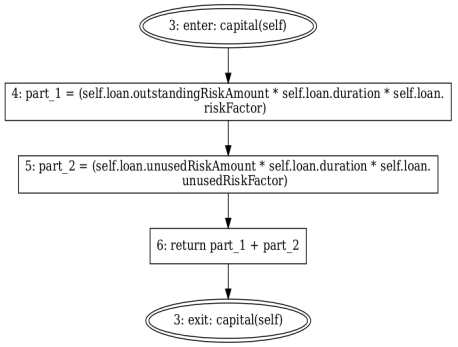


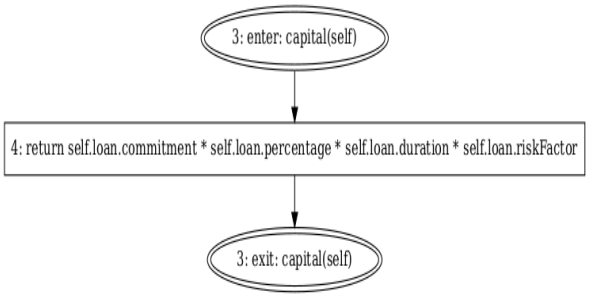
Figura 19 – *Call graph* após a refatoração



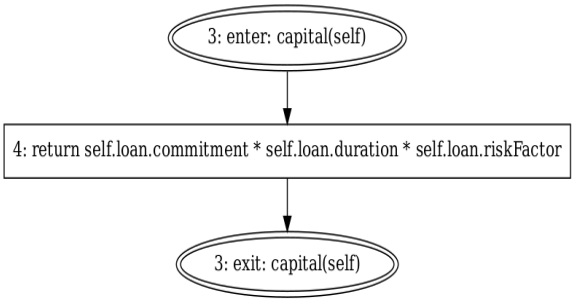
(a) Strategy



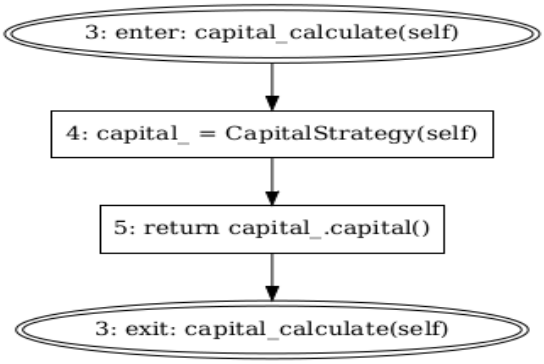
(b) StrategyTermLoan



(c) StrategyAdvised



(d) StrategyRevolver



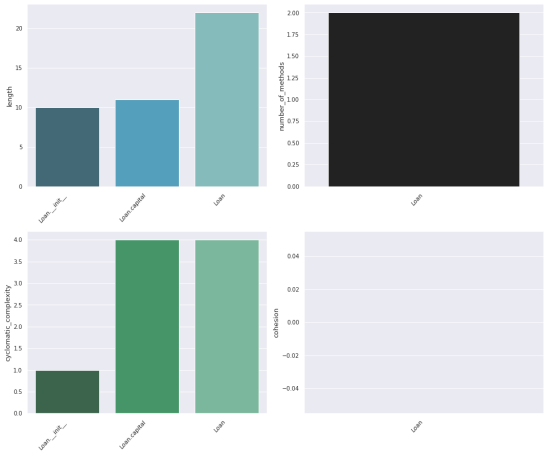
(e) Loan

Figura 20 – *Flow graph* após a refatoração

5.2.4.0.2 Métricas e Resultados

As métricas antes da refatoração apresentam um alto valor de complexidade ciclomática e linhas de código no método calculate. Isto posto, é possível que as operações a serem realizadas sejam substituir lógica condiciional por polimorfismo ou substituir lógica condicional por strategy, como é mostrado nas figuras 21a e 21b.

As métricas após a refatoração apresentam apenas linhas de código no método ___init___, com um valor elevado, pelo motivo de ser ali onde os atributos são declarados. Logo, mesmo que a aplicação mostre que este as operações anteriores continuem com 50%



(a) Gráfico de métricas antes da refatoração

replace conditional with polymorphism

	item	percent	match
0	Loan.__init__	0.5	[length]
1	Loan.capital	1.0	[cyclomatic_complexity, length]

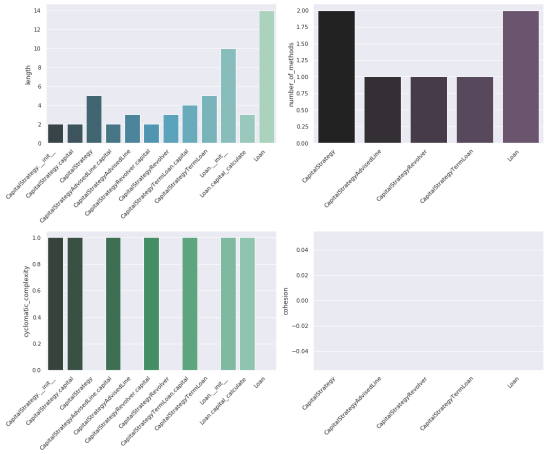
replace conditional logic with strategy

	item	percent	match
0	Loan.__init__	0.5	[length]
1	Loan.capital	1.0	[cyclomatic_complexity, length]

(b) Tabela de métricas antes da refatoração

Figura 21 – Resultados antes da refatoração (Fonte: Autor)

de correspondência nesse método, não é um contexto de refatoração, como é mostrado nas figuras 22a e 22b.



(a) Gráfico de métricas após a refatoração

replace conditional with polymorphism

	item	percent	match
0	CapitalStrategy.__init__	0.0	[]
1	CapitalStrategy.capital	0.0	[]
2	CapitalStrategyAdvisedLine.capital	0.0	[]
3	CapitalStrategyRevolver.capital	0.0	[]
4	CapitalStrategyTermLoan.capital	0.0	[]
5	Loan.__init__	0.5	[length]
6	Loan.capital_calculate	0.0	[]

replace conditional logic with strategy

	item	percent	match
0	CapitalStrategy.__init__	0.0	[]
1	CapitalStrategy.capital	0.0	[]
2	CapitalStrategyAdvisedLine.capital	0.0	[]
3	CapitalStrategyRevolver.capital	0.0	[]
4	CapitalStrategyTermLoan.capital	0.0	[]
5	Loan.__init__	0.5	[length]
6	Loan.capital_calculate	0.0	[]

(b) Tabela de métricas após a refatoração

Figura 22 – Resultados após da refatoração (Fonte: Autor)

5.2.5 Implementação

Da proposta inicial da ferramenta foram concluídos três módulos a manipulação da entrada, que consiste na captura do diretório e na obtenção dos arquivos do sistema a ser analisado. A extração das representações por meio do *flow graph* e *call graph* e a análise dos sistema por meio das métricas.

O módulo core, responsável pela integração do sistema foi parcialmente desenvolvido, pois o core é responsável pela comunicação entre os módulos e nem todos foram finalizados.

A aplicação das operações não foi desenvolvida e por conseguinte o módulo de manipulação de saída também não. A saída foi planejada como sendo o sistema após a aplicação das operações de refatoração, no entanto, por tempo e complexidade este módulo não foi implementado. Sendo a saída a exibição de tabelas com a correspondência do contexto achado com o contexto esperado.

6 Considerações Finais

Com base na pesquisa realizada é possível afirmar que existem alguns padrões presentes em código que podem caracterizar uma situação de refatoração para potencializar a variabilidade de um sistema. Logo, existem formas de mapear isso por meio de métricas e representações que podem ser utilizadas para comparar com outras situações.

É evidenciado nos resultados que dado um contexto é possível indicar algumas técnicas de refatoração para aprimorar a variabilidade em sistemas de software por meio de uma mudança de design, aplicando operações referentes a *design patterns*.

As métricas selecionadas não tem caracter discriminatório exato, uma vez que alguns contextos requeridos paras as operações não são únicos, podendo haver mais de uma operação aplicável. Portanto, para continuidade em trabalhos futuros se faz necessário a adaptação de algumas métricas e o aprimoramento das representações.

As bibliotecas utilizadas foram exploradas para verificar a viabilidade de uso e prováveis limitações. A biblioteca *pycfg* tem uma limitação informada pelo próprio mantenedor que é o não reconhecimento de exceções. Além disso, como é uma biblioteca destinada a linguagem com paradigma estruturado, não consegue reconhecer métodos com mesmo nome mas de diferentes classes e por este motivo foram feitas algumas adaptações em código para que a ferramenta.

A de inferência de tipos, *pytype*, tem uma limitação, também informada pelo mantenedor que diz que a inferência não é totalmente precisa. Dificultando a observação de alguns padrões e cálculo de métricas e a representação do grafo de chamadas.

Como o python utiliza uma programação com tipos dinâmicos não foi possível aferir métricas relacionadas a acoplamento, uma vez que dada um a interface comum, qualquer tipo de objeto pode ser utilizado. Uma forma de aferir tais métricas é determinando isso em tempo de execução salvando as referências para cada um dos tipos referenciados.

Quanto aos trabalhos futuros, a análise dos atributos relacionados a variabilidade podem ser mais aprofundados para um resultado mais minuncioso. Além disso, é possível a realização de novos funcionalidades, como a aplicação das operações na aplicação após às análises e a utilização de otimização dos resultados utilizando SBSE ou algumas técnicas de *machine learning*.

Referências

AL-JA'AFER, J.; SABRI, K. Metrics for object oriented design (mood) to assess java programs. *King Abdullah II school for information technology, University of Jordan, Jordan*, 2004. Citado 5 vezes nas páginas 74, 90, 92, 93 e 94.

APEL, S. et al. *Feature-oriented software product lines*. [S.l.]: Springer, 2016. Citado 6 vezes nas páginas 6, 20, 21, 22, 23 e 24.

BRUMLIK, J.; VANNIN, T. M. The Use and Limitations of Static-Analysis Tools to Improve Software Quality. *J.Occup.Med.*, v. 12, n. 0096-1736 (Print), p. 308–314, 1970. Citado na página 40.

CHHIKARA, A.; CHHILLAR, R. Analyzing the complexity of java programs using object-oriented software metrics. *International Journal of Computer Science Issues (IJCSI)*, Citeseer, v. 9, n. 1, p. 364, 2012. Citado 2 vezes nas páginas 73 e 75.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, IEEE, v. 20, n. 6, p. 476–493, 1994. Citado 3 vezes nas páginas 60, 70 e 76.

CLEMENTS, P.; NORTHROP, L. *Software product lines*. [S.l.]: Addison-Wesley Boston, 2002. Citado na página 17.

DHILLON, B. S. *Engineering maintenance: a modern approach*. [S.l.]: cRc press, 2002. Citado 2 vezes nas páginas 16 e 39.

FAIRLEY, R. E. Tutorial: Static Analysis and Dynamic Testing of Computer Software. *Computer*, v. 11, n. 4, p. 14–23, 1978. ISSN 00189162. Citado 2 vezes nas páginas 39 e 40.

FIORAVANTI, F.; NESI, P. A method and tool for assessing object-oriented projects and metrics management. *Journal of Systems and Software*, Elsevier, v. 53, n. 2, p. 111–136, 2000. Citado 4 vezes nas páginas 63, 65, 67 e 69.

FOWLER, M. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999. Citado 4 vezes nas páginas 27, 28, 29 e 30.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2. Citado 7 vezes nas páginas 6, 16, 17, 23, 24, 25 e 26.

GENERO, M.; PIATTINI, M.; CALERO, C. A survey of metrics for uml class diagrams. *Journal of object technology*, v. 4, n. 9, p. 59–92, 2005. Citado 4 vezes nas páginas 79, 85, 86 e 87.

GIL, A. C. Como elaborar projetos de pesquisa. *São Paulo*, v. 5, n. 61, p. 16–17, 2002. Citado na página 18.

GURP, J. V.; BOSCH, J.; SVAHNBERG, M. On the notion of variability in software product lines. In: IEEE. *Proceedings Working IEEE/IFIP Conference on Software Architecture*. [S.l.], 2001. p. 45–54. Citado 2 vezes nas páginas 16 e 20.

HARMAN, M.; JONES, B. F. Search-based software engineering. *Information and software Technology*, Elsevier, v. 43, n. 14, p. 833–839, 2001. Citado 2 vezes nas páginas 17 e 36.

HARMAN, M. et al. Search based software engineering: Techniques, taxonomy, tutorial. In: *Empirical software engineering and verification*. [S.l.]: Springer, 2010. p. 1–59. Citado 5 vezes nas páginas 6, 17, 36, 37 e 38.

HENDERSON-SELLERS, B. *Object-oriented metrics: measures of complexity*. [S.l.]: Prentice-Hall, Inc., 1995. Citado na página 82.

HITZ, M.; MONTAZERI, B. Measuring coupling and cohesion in object-oriented systems. In: *Proceedings of International Symposium on Applied Corporate Computing*. [S.l.: s.n.], 1995. p. 25–27. Citado na página 81.

IBRAHIM, A. A. E.; KAMEL, A.; HASSAN, H. Object oriented metrics and quality attributes: A survey. In: *Proceedings of the 10th International Conference on Informatics and Systems*. [S.l.: s.n.], 2016. p. 312–319. Citado na página 83.

KERIEVSKY, J. *Refactoring to Patterns*. [S.l.]: Pearson Higher Education, 2004. ISBN 0321213351. Citado 7 vezes nas páginas 16, 27, 29, 31, 32, 33 e 50.

KIM, H.; BOLDYREFF, C. Developing software metrics applicable to uml models. In: *6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*. [S.l.: s.n.], 2002. p. 1–10. Citado na página 72.

KLEINBERG, J.; TARDOS, E. *Algorithm design*. [S.l.]: Pearson Education India, 2006. Citado 3 vezes nas páginas 34, 35 e 40.

KRUEGER, C. W. Software reuse. *ACM Computing Surveys (CSUR)*, ACM, v. 24, n. 2, p. 131–183, 1992. Citado na página 20.

KUROSE, J. F.; ROSS, K. W. Redes de computadores e a internet. *Uma nova*, 2006. Citado 2 vezes nas páginas 6 e 35.

LARMAN, C. *Applying UML and patterns: an introduction to object oriented analysis and design and interative development*. [S.l.]: Pearson Education India, 2012. Citado 2 vezes nas páginas 76 e 81.

LI, W.; HENRY, S. Maintenance metrics for the object oriented paradigm. In: IEEE. *[1993] Proceedings First International Software Metrics Symposium*. [S.l.], 1993. p. 52–60. Citado na página 64.

MCCABE, T. J. A complexity measure. *IEEE Transactions on software Engineering*, IEEE, n. 4, p. 308–320, 1976. Citado na página 64.

MORESI, E. et al. Metodologia da pesquisa. *Brasília: Universidade Católica de Brasília*, v. 108, p. 24, 2003. Citado na página 18.

NGUYEN, V. et al. A sloc counting standard. In: CITESEER. *Cocomo ii forum*. [S.l.], 2007. v. 2007, p. 1–16. Citado na página 60.

PARNAS, D. L. Designing software for ease of extension and contraction. *IEEE transactions on software engineering*, IEEE, n. 2, p. 128–138, 1979. Citado na página 20.

PIZKA, M.; DEISSENBÖCK, F. How to effectively define and measure maintainability. *SMEF*, 2007. Citado na página 16.

RFLEMING. *Static Analysis*. 2018. Disponível em: <https://www.grammatech.com/products/source-code-analysis>. Citado 2 vezes nas páginas 6 e 40.

RIEGER, M.; DUCASSE, S.; LANZA, M. Insights into system-wide code duplication. In: IEEE. *11th Working Conference on Reverse Engineering*. [S.l.], 2004. p. 100–109. Citado na página 95.

SCHACH, S. R.; TOMER, A. Development/maintenance/reuse: software evolution in product lines. In: *Software Product Lines*. [S.l.]: Springer, 2000. p. 437–450. Citado na página 17.

SOMMERVILLE, I. *Engenharia de software*. [S.l.]: PEARSON BRASIL, 2011. ISBN 9788579361081. Citado na página 39.

SVAHNBERG, M.; GURP, J. V.; BOSCH, J. A taxonomy of variability realization techniques. *Software: Practice and experience*, Wiley Online Library, v. 35, n. 8, p. 705–754, 2005. Citado na página 20.

Apêndices

APÊNDICE A – Catálogo de Métricas

A.1 Source Lines of Code (SLOC)

Geralmente, quanto maior o tamanho do código maior é sua complexidade e maior é a dificuldade de manter tal sistema. A métrica *source lines of code*, ou simplesmente *SLOC*, é a contagem de linhas de código de um determinado elemento, seja ele um subsistema, módulo, classe ou método (NGUYEN et al., 2007).

A.1.1 Fator de qualidade

A complexidade do sistema aumenta de acordo com a quantidade de código inserido nele, logo, a alternância da métrica SLOC deve implicar em uma alternância no seu grau de manutenibilidade.

A.1.2 Cálculo

Segundo (NGUYEN et al., 2007), não existe uma forma universal de contagem dessa métrica, muito pelo fato dela ser utilizada de insumo para alguns sistemas de precificação de software como o COCOMO, SLIM, SEER-SEM, e PriceS. Nesses casos o *SLOC* pode envolver dois conceitos SLOC físico que é representado pela quantidade de linhas que não contêm apenas espaços ou comentários e SLOC lógico que pode ser medido pela quantidade de instruções lógicas contidas no código.

$$SLOC = Linhasdecódigofonte$$

A.1.3 Aplicação da fórmula

Será utilizado, para fins de cálculo o SLOC físico, feito pela contagem absoluta do início ao fim do código. Como a fórmula é simples, bastando a realização da contagem absoluta de linhas, o trecho de código presente em 1 tem

$$SLOC = 10$$

A.2 Weighted Methods per Class (WMC)

A métrica *Weighted Methods per Class* consiste no somatório das complexidades dos métodos de uma classe. Segundo (CHIDAMBER; KEMERER, 1994), o número de

```
def calculate_tax(value):
    if value > 5000:
        tax = value * 0.2
    elif value > 2500:
        tax = value * 0.15
    elif value 1000:
        tax = value * 0.1
    else:
        tax = 0

    print(f'Tax: {tax}')
```

Listing 1 – Código fonte - LOC

métodos e a complexidade destes é um preditor de esforço e tempo necessários para desenvolver e manter uma classe, também é maior o impacto aos possíveis filhos que herdarão todos os métodos da classe pai e geralmente estas classes tendem a ser de limitada reutilização por serem provavelmente mais relacionadas a um aplicativo específico.

A.2.1 Fator de qualidade

Os fatores de qualidade que podem ser identificados por meio desta métrica são a reusabilidade e a manutenibilidade. Uma vez que a coesão um WMC baixo pode indicar uma coesão maior e um grau de manutenibilidade e reutilização maior.

A.2.2 Cálculo

Considere uma classe *C1*, com os métodos *M1*, ... *Mn* que estão definidos na classe. Seja c1, ... *cn* seja a complexidade dos métodos. Então:

$$WMC = \sum_{i=1}^{n} ci$$

Se todas as complexidades do método forem consideradas unitárias, então WMC = n , ou seja, WMC é igual ao número de métodos da classe.

A.2.3 Aplicação da fórmula

A Figura 23 apresenta um diagrama de classes com três métodos, como a métrica WMC não define qual critério será utilizada para o calculo da complexidade, suponha que a complexidade do *methodA* seja 1, do *methodB* seja 5 e do *methodC* seja 3, então:

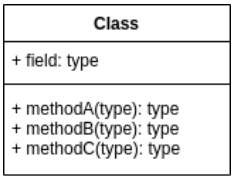


Figura 23 – Diagrama WMC (Fonte: Autor)

$$WMC = \sum_{i=1}^{n} ci$$
$$WMC = \sum(1, 5, 3)$$
$$WMC = 9$$

A.3 Average Number of Parameters per Method (ANPM)

A métrica *Average Number of Parameters per Method* tem o objetivo de aferir a média de parâmetros por método em uma determinada classe. Ou seja, dado uma classe é pego a soma da quantidade de parâmetos por método e este valor é dividido com NOM.

A.3.1 Fator de qualidade

Não existe mínimo ou máximo, no entanto um grande números de parâmetros pode indicar que a classe está com mais responsabilidade do que o necessário, o que implica em baixa coesão na classe.

A.3.2 Cálculo

$$ANPM = \frac{\sum_{i=1}^{n} P_m(i)}{\sum_{i=1}^{M_i}}$$

Ou

$$ANPM = \frac{\sum_{i=1}^{n} P_m(i)}{NOM}$$

Onde:

- Pm é a quantidade de paramêtros do método i
- Mi representa os métodos da classe

A.3.3 Aplicação da fórmula

A.4 Total Number of Methods (TNM)

A métrica *Total Number of Methods* é uma métrica relacionada aos métodos de uma classe que indica todos os métodos da classe, sejam eles declarados na própria classe ou herdados (FIORAVANTI; NESI, 2000).

A.4.1 Fator de qualidade

O Fator de Qualidade afetado por essa métrica é o atributo de dimensionamento, que mede o tamanho de uma unidade de design. Quanto maior essa métrica, Quanto maior é essa métrica, maior é o atributo de dimensionamento.

A.4.2 Cálculo

$$TNM = \sum_{i=1}^{n} M_i + \sum_{i=1}^{n} M_d$$

Onde

- Md representa os atributos declarados;
- Mi os atributos herdados.

A.4.3 Aplicação da fórmula

A Figura 24 é um diagrama com três classes. Onde, a ClasseA tem três métodos, a ClasseB tem dois métodos e a ClasseC tem dois métodos declarados e um herdado. Logo:

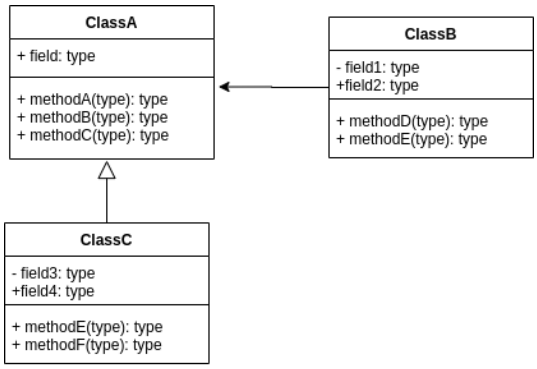


Figura 24 – Diagrama TNM (Fonte: Autor)

$$TNM(ClasseA) = 0 + 3 = 3$$

$$TNM(ClasseB) = 0 + 2 = 2$$

$$TNM(ClasseC) = 3 + 2 = 5$$

A.5 Number of Methods (NOM)

Essa métrica está relacionada a coesão e quanto maior é seu valor mais difícil é uma classe estar coesa, uma vez que é possível deduzir que a classe está com muitas responsabilidades indicando que a interface da classe está complexa (LI; HENRY, 1993).

A.5.1 Fator de qualidade

Assim com em *WMC*, um número de métodos elevado, geralmente, indica uma dificuldade de reúso e de manutenção elevada. Portanto, os fatores relacionados a essa métrica são reutilização e manutenibilidade.

A.5.2 Cálculo

O cálculo da métrica de número de métodos é simples, é contagem absoluta do número de métodos de uma classe.

$$NOM = Número de métodos$$

A.5.2.1 Aplicação

Assim como LOC o números de métodos de uma classe é uma contagem simples e absoluta. O trecho de código apresentado em 5 exemplifica uma classe com

$$NOM = 3$$

A.6 Cyclomatic Complexity (CC)

Uma das métricas para cálculo de complexidade de um trecho de código fonte é a complexidade ciclomática, que representa a quantidade de caminhos de tal trecho de código (MCCABE, 1976).

A.6.1 Fator de qualidade

A complexidade do sistema aumenta de acordo com a complexidade de código inserido nele, logo, a alternância da métrica CC deve implicar em uma alternância no seu grau de manutenibilidade.

```
1  class Shape:
2  def __init__(self, legend):
3      self.legend = legend
4
5  def draw(self):
6      print('Draw shape')
7
8  def calculate_area(self):
9      print('Area')
10
11 def calculate_perimeter(self):
12     print('Perimeter')
```

Listing 2 – Código fonte - NOM

A.6.2 Cálculo

A complexidade ciclomática pode ser calculada de acordo com essas ramificações. O número de caminhos que são criados por estruturas condicionais e estruturas de repetição entram no cálculo dessa métrica.

$$M = E - N + 2$$

Logo, a fórmula representa o cálculo dessa métrica, onde E é o número de arestas e N o número de nós.

A.6.3 Aplicação da fórmula

A Figura 25 e 3 apresentam um código onde o número de arestas é de 10 e o número de nós é 9, totalizando uma complexidade ciclomática de 3.

$$M = E - N + 2$$

$$M = 10 - 9 + 2$$

$$M = 3$$

A.7 Total Number of Attributes (TNA)

A métrica *Total Number of Attributes* é uma métrica relacionada aos atributos de uma classe que indica todos os atributos da classe, sejam eles declarados na própria classe ou herdados (FIORAVANTI; NESI, 2000).

```python
def check_number():
    number = 0
    while number != 1:
        number = int(inpit())
        if number % 2 == 0:
            is_even = True
        else:
            is_even = False

        print(f'Is even ? {is_even}')
```

Listing 3 – Código fonte - CC

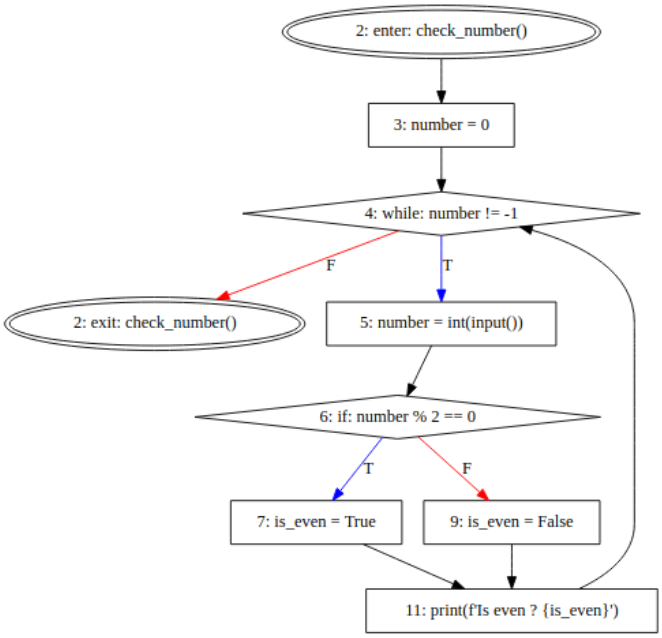


Figura 25 – Flow graph

A.7.1 Fator de qualidade

O Fator de Qualidade afetado por essa métrica é o atributo de dimensionamento, que mede o tamanho de uma unidade de design. Quanto maior essa métrica, Quanto maior é essa métrica, maior é o atributo de dimensionamento.

A.7.2 Cálculo

$$TNA = \sum_{i=1}^{n} A_i + \sum_{i=1}^{n} A_d$$

Onde

- Ad representa os atributos declarados;
- Ai os atributos herdados.

A.7.3 Aplicação da fórmula

A Figura 26 é um diagrama com cinco classes. Onde, a ClasseA tem um atributo, a ClasseB tem dois atributos declarados e um herdado, a ClasseC tem três atributos e um herdado, a ClasseD tem um atributo declarado e três herdados e a ClasseE tem dois atributos declarados e quatro atributos herdados. Logo:

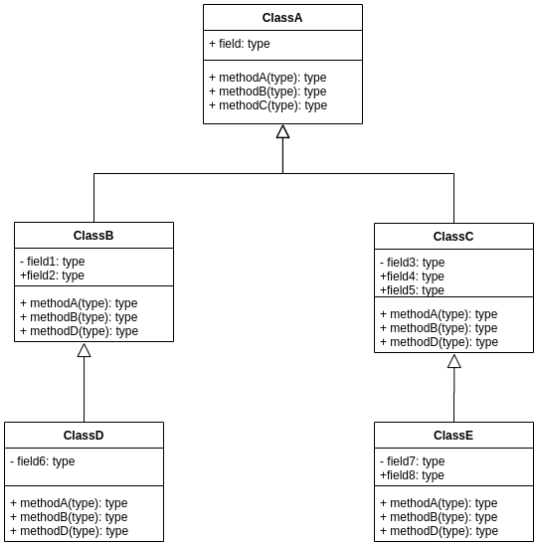


Figura 26 – Diagrama TNA (Fonte: Autor)

$$TNA(ClasseA) = 0 + 1 = 1$$

$$TNA(ClasseB) = 1 + 2 = 3$$

$$TNA(ClasseC) = 1 + 3 = 4$$

$$TNA(ClasseD) = 3 + 1 = 4$$

$$TNA(ClasseE) = 4 + 2 = 6$$

A.8 Number of Attributes (NumAttr)

A métrica *Number of Attributes* é uma métrica relacionada aos atributos de uma classe que indica todos os atributos da classe e, ao contrário da métrica TNA, apenas considera os atributos declarados na própria classe (FIORAVANTI; NESI, 2000).

A.8.1 Fator de qualidade

Uma classe com muitos atributos pode indicar a presença de coesão coincidente e requer mais decomposição, para melhor gerenciar a complexidade do modelo. Se não houver atributos, deve ser dada uma atenção séria à semântica da classe, se de fato houver alguma. Isso pode ser um utilitário de classe em vez de uma classe.

A.8.2 Cálculo

$$NumAttr = \sum_{i=1}^{n} A_i$$

Onde Ai representa os atributos da classe.

A.8.3 Aplicação da fórmula

A Figura 27 é um diagrama com cinco classes. Onde, a ClasseA tem um atributo, a ClasseB tem dois atributos , a ClasseC tem três atributos, a ClasseD tem um atributo e a ClasseE tem dois atributos declarados. Logo:

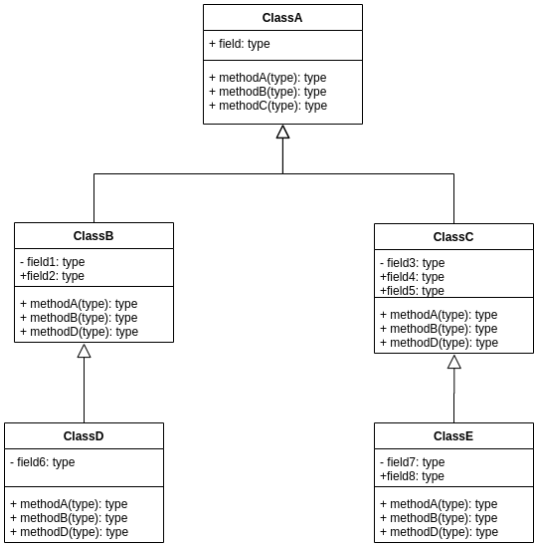


Figura 27 – Diagrama NumAttr (Fonte: Autor)

$$TNA(ClasseA) = 1$$

$$TNA(ClasseB) = 2$$

$$TNA(ClasseC) = 3$$

$$TNA(ClasseD) = 1$$

$$TNA(ClasseE) = 2$$

A.9 Total Number of Classes (TNC)

A métrica *Total Number of Classes* é uma métrica que conta, de forma absoluta, a quantidade de classes no sistema de software (FIORAVANTI; NESI, 2000).

A.9.1 Fator de qualidade

A referência bibliográfica da métrica não atribui nenhum fator de qualidade específico.Porém sua utilização está relacionada a compreensão do códifo, sua análise e a facilidade de modificação.

A.9.2 Cálculo

$$TNC = \sum_{i=1}^{n} C_i$$

Onde Ci representa as classes do sistema.

A.9.3 Aplicação da fórmula

A Figura 28 é um diagrama com sete classes, sendo elas: ClasseA, ClasseB ClasseC, ClasseD, ClasseE, ClasseF e ClasseG. Logo:

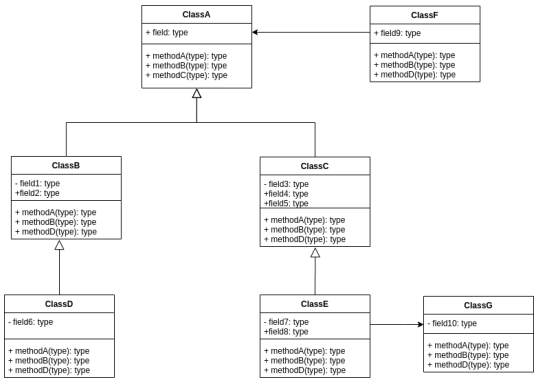


Figura 28 – Diagrama TNC (Fonte: Autor)

$$TNC = 7$$

A.10 Size2

A métrica *Size2* calcula todos os estados e ações de uma classe, ou seja, é a soma de todos métodos e atributos de uma classe (FIORAVANTI; NESI, 2000).

A.10.1 Fator de qualidade

Quanto menor o Size2 pode se considerar que é uma classe mais simples. E quando essa métrica retorna valores muito alto é um indicativo que a classe deve ser revista, pois apresenta um alto grau de complexidade. É uma classe que está fazendo muitas ações e tendo muito atributos, o que torna um ponto de uma possível refatoração.

A.10.2 Cálculo

$$Size2 = \sum_{i=1}^{n} M_i + \sum_{i=1}^{n} A_i$$

Onde:

- Ai representa os atributos
- Mi representa os métodos

A.10.3 Aplicação da fórmula

A Figura 28 é um diagrama com duas classes. A ClasseA tem um atributo e três métodos e a ClasseB tem dois atributos e dois métodos. Logo:

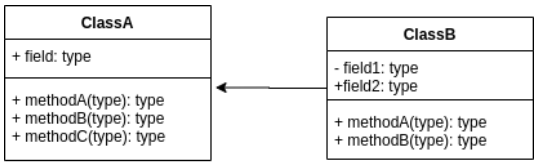


Figura 29 – Diagrama Size2 (Fonte: Autor)

$$Size2(ClasseA) = 1 + 3 = 4$$

$$Size2(ClasseB) = 2 + 2 = 4$$

A.11 Depth of Inheritance (DIT)

A métrica *Depth of Inheritance* busca medir a profundidade de uma classe em uma árvore de herança, ou seja, em qual nível, dado um nó raiz em que não existe hierarquia (0) (CHIDAMBER; KEMERER, 1994).

A.11.1 Fator de qualidade

Quanto maior o valor de DIT, mais difícil é para manter uma classe.Isso acontece porque quanto maior o DIT que a classe tem, mais ela acessa propriedades de outras superclasses, e isso pode acabar violando o encapsulamento.

A.11.2 Cálculo

Essa métrica pode assumir um valor inteiro,variando de 0 a N. O cálculo é realizado por meio de uma contagem da posição que esta classe pertence em uma árvore de heranças.

$$DIT = numberoflevels$$

A.11.3 Aplicação da fórmula

Considerando que uma classe tem a possibilidade de multiplas heranças, a Figura 30 apresenta seis classes, onde as classes ClasseA, ClasseF e ClasseD não herdam de nenhuma outra, as classes ClasseB e ClasseC herdam da ClasseA e a ClasseE herda da ClasseD e da ClasseD. Logo:

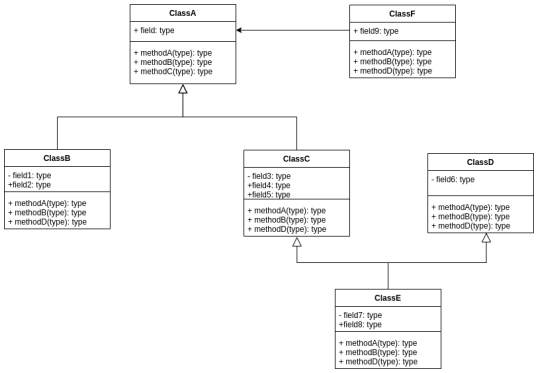


Figura 30 – Diagrama dit (Fonte: Autor)

$$DIT(ClasseA) = 0$$

$$DIT(ClasseB) = 1$$

$$DIT(ClasseC) = 1$$

$$DIT(ClasseD) = 0$$

$$DIT(ClasseE) = 2$$

$$DIT(ClasseF) = 0$$

A.12 Number of Associations (NASC)

A métrica *Number of Associations* calcula a quantidade de relacionamentos, de forma estática, seja ela relacionamentos por associação, composição ou agregação, não inclui herança (KIM; BOLDYREFF, 2002).

A.12.1 Fator de qualidade

Esta métrica está fortemente associada ao acoplamento entre a classe em questão e as demais. Quanto maior esse número mais acoplada está a classe e quanto menor esse número menos acoplado está a classe.

A.12.2 Cálculo

$$NASC = \sum Association$$

A.12.3 Aplicação da fórmula

A Figura 31 é um diagrama com quatro classes. A ClasseA tem associação com uma classe e agregação com outra, a ClasseB tem associação com uma classe, a classeC não tem associações e a ClasseD possui agregação com a ClasseA. Logo:

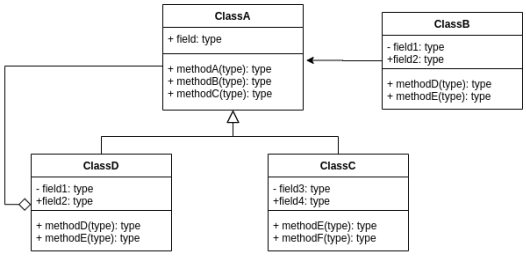


Figura 31 – Diagrama NASC (Fonte: Autor)

$$NASC(ClasseA) = 2$$
$$NASC(ClasseB) = 1$$
$$NASC(ClasseC) = 0$$
$$NASC(ClasseD) = 1$$

A.13 Response for a Class (RFC)

A *Response for a Class* é uma métrica orientada a objetos que mostra a interação dos métodos da classe com outros métodos. O valor é calculado pelo somatório de todos

```
class Circle:
    def display(self):
        self.calculate_area()
        self.calculate_radius()
        draw_image = DrawImage(self)
        draw_image.show()

    def calculate_area(self):
        print('Area')

    def calculate_radius(self):
        print('Radius')

class DrawImage:
    def show(self):
        print('Show')
```

Listing 4 – Código fonte - NOM

os métodos daquela classe, e todos os métodos chamados diretamente por essa classe (CHHIKARA; CHHILLAR, 2012).

A.13.1 Fator de qualidade

Uma classe com RFC alto pode ser uma classe com uma grande quantidade de métodos ou uma classe bastante dependente das demais classes do sistema. Tendo um valor alto de RFC constatado em uma classe é uma classe que deve ser revista para uma possível refatoração, pois indicar baixa coesão e alto acoplamento.

A.13.2 Cálculo

A resposta definida para a classe pode ser expressa como:

$$RFC = \{M\} \cup_{alli} \{R_i\}$$

onde { Ri } é o conjunto de métodos chamados pelo método i e { M } é o conjunto de todos os métodos da classe.

A.13.3 Aplicação da fórmula

$$RFC(Circle) = \{Circle :: display, Circle :: calculate\_area, Circle :: calculate\_radius\}$$

$$\cup$$

$$\{Circle :: calculate\_area, Circle :: calculate\_radius, DrawImage :: draw\}$$

$$RFC(Circle) = \{Circle :: display, Circle :: calculate\_area,$$
$$Circle :: calculate\_radius, DrawImage :: draw\}$$
$$RFC(Circle) = 4$$

A.14 Coupling Factor (COF)

A métrica *Coupling Factor* é uma métrica de acoplamento entre as classes. É a razão entre o número de pares acoplados e o número possível de pares (AL-JA'AFER; SABRI, 2004).

A.14.1 Fator de qualidade

A relação cliente-servidor entre duas classes acontece quando uma determinada classe acessa atributos ou métodos de outra classe. Essa relação caracteriza acoplamento entre as classes

A.14.2 Cálculo

$$COF = \frac{\sum_{i=1}^{TC}[\sum_{j=1}^{TC} isClient(C_i, C_j)]}{TC^2 - TC}$$

Onde:

$$isClient(Cc, Cs) = \begin{cases} 1, & se\ (C_c \Rightarrow C_s) \wedge (C_c \neq C_s) \\ 0, & senão \end{cases}$$

A.14.3 Aplicação da fórmula

A Figura 32 é um diagrama com quatro classes. Onde a classe Cliente tem um cliente, a classe Venda não tem nenhuma, a classe Item tem um cliente e a classe Produto tem um cliente. Logo:

$$COF = \frac{1+0+1+1}{4^2-4}$$
$$COF = \frac{3}{12}$$
$$COF = 0.25$$

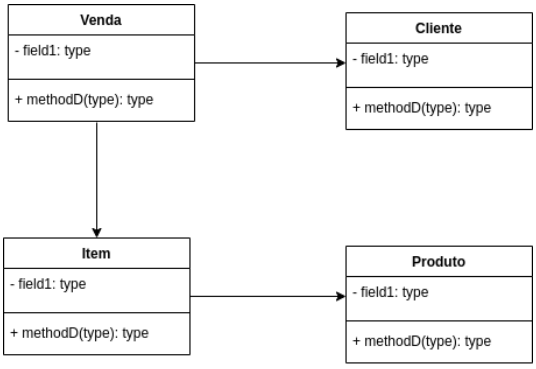


Figura 32 – Diagrama COF (Fonte: Autor)

A.15 Data Abstraction Coupling (DAC)

A métrica *Data Abstraction Coupling* mede a complexidade do acoplamento causada por *Abstract Data Types (ADTs)*. Esta métrica está relacionada ao acoplamento entre classes que representam um aspecto importante do projeto orientado a objeto, uma vez que o grau de reutilização, o esforço de manutenção e teste para uma classe são decisivamente influenciados pelo nível de acoplamento entre classes. Acoplamento de Abstração de Dados (DAC) é o número de atributos em uma classe que tenha outras classes como seu tipo (CHHIKARA; CHHILLAR, 2012).

A.15.1 Fator de qualidade

Quanto mais ADTs uma classe tiver, mais complexo será o acoplamento dessa classe com outras classes. Portanto quanto maior o número DAC maior será o acoplamento e quanto menor o DAC menos acoplado está essa classe com outras classes.

A.15.2 Cálculo

$$DAC = \sum_{i=1}^{n} isADT(A_i)$$

Onde:

$$isADT(A_i) = \begin{cases} 1, & se\ (A_i\ for\ um\ atributo\ ADT \\ 0, & senão \end{cases}$$

A.15.3 Aplicação da fórmula

A Figura 33 é um diagrama com cinco classes. Onde as classes Room, ColorRoom e CurtainsRoom não tem nenhuma ADT, a classe BaseRoom tem uma ADT e a classe DecoratorRoom tem duas ADT's a classe. Logo:

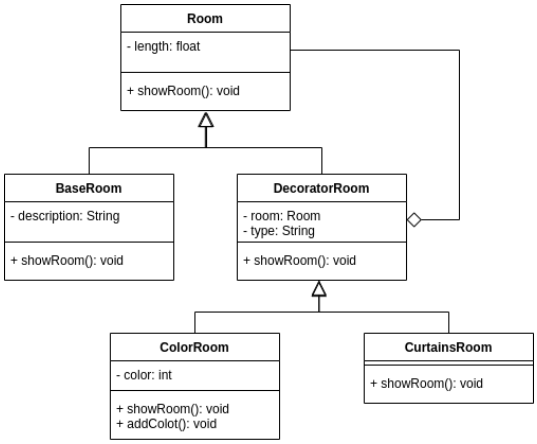


Figura 33 – Diagrama DAC (Fonte: Autor)

$$DAC(Room) = 0$$

$$DAC(ColorRoom) = 0$$

$$DAC(CurtainsRoom) = 0$$

$$DAC(BaseRoom) = 1$$

$$DAC(DecoratorRoom) = 2$$

A.16 Coupling between Objects (CBO)

O acoplamento pode ser entendido como grau de interdependência entre módulos de um software. Isto é, indica quanto um módulo utiliza ou é utilizado por outro módulo. Geralmente, quanto mais baixo o nível de acoplamento mais bem estruturado é o sistema (LARMAN, 2012). Existem diversas métricas para cálculo do acoplamento. Contagens relacionados à associações, agregações, composições e hierarquias são feitos para obter o valor desta métrica. *Coupling between Objects (CBO)* é uma das métricas comumente utilizadas em OO.

A.16.1 Fator de qualidade

O acoplamento grande entre os objetos pode gerar uma dependência grande entre módulos, componentes e classes e por isso dificulta seu reúso e sua manutenção. Portanto os fatores referentes a está métrica são a reutilização e a manutenibilidade.

A.16.2 Cálculo

Dada uma classe A, Ai é acoplada a A se a classe Ai mantêm alguma referência para A ou o inverso, em outras palavras Ai é cliente de A ou A é cliente de Ai. Onde A é diferente de Ai (CHIDAMBER; KEMERER, 1994).

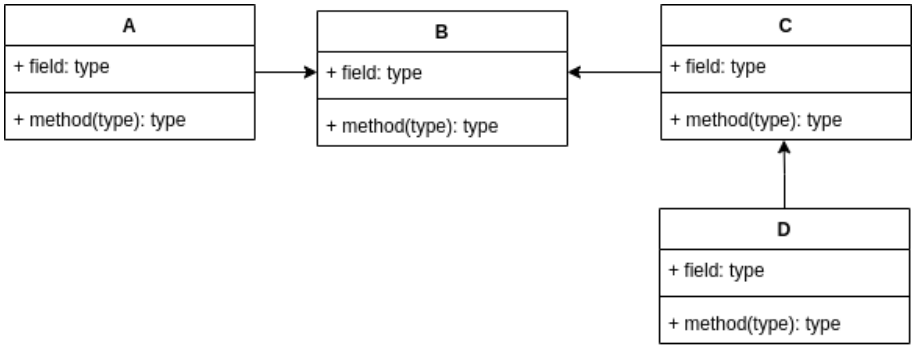


Figura 34 – Exemplo acoplamento

$$CBO(A) = \sum_{i=1}^{n} Depends(A, Ai)$$

A.16.3 Aplicação da fórmula

Na Figura 34 temos a seguinte configuração para a métrica *CBO*:

- *CBO(A) = 1*, já que *A* referencia *B*;
- *CBO(B) = 2*, uma vez que *A* referencia *B* e *B* referencia *C*;
- *CBO(C) = 2*, visto que *C* referencia *B* e *D* referencia *C*;
- *CBO(D) = 1*, dado que *D* referencia *C*.

A.17 Message Passing Coupling (MPC)

A métrica *Message Passing Coupling* mede a quantidade de menssagens trocadas entre classe. Valores altos indicam uma dependência grande de outros objetos e um valor baixo, ao contrário, indica uma dependência baixa de outras classes.

A.17.1 Fator de qualidade

Essa métrica ajuda a identificar classes com níveis de acoplamento mais elevados, que é um indicativo de uma classe complexa. Essa métrica auxilia a identificar classes que serão difíceis de manter e entender. Portanto, essa métrica auxilia a análise da capacidade de manutenção de uma classe.

A.17.2 Cálculo

$$MPC = sum_{i=1}^{n} C_h(i)$$

```python
class Circle:
    def display(self):
        self.calculate_area()
        self.calculate_radius()
        draw_image = DrawImage(self)
        draw_image.show()

    def calculate_area(self):
        print('Area')

    def calculate_radius(self):
        print('Radius')

class DrawImage:
    def show(self):
        print('Show')
```

Listing 5 – Código fonte - NOM

Onde Ch(i) indica a chamada de um método de outra classe.

A.17.3 Aplicação da fórmula

$$MPC(Circle) = 1$$

$$MPC(DrawImage) = 0$$

A.18 Number of Dependencies In (NDepIn)

A métrica *Number of Dependencies In* contabiliza o número de dependentes de uma determinada classe, ou seja, mede o quanto outras classes utilizam uma classe específica citegenero2005survey.

A.18.1 Fator de qualidade

O fator de qualidade afetado por essa métrica é o acoplamento, que é um conceito orientado a objetos que se refere a como as classes ou módulos são relacionados ou dependem uns dos outros. Ele mostra o grau de interdependência entre as classes com a força das relações entre os módulos.

A.18.2 Cálculo

$$NDepIn(C) = \sum_{i=1}^{n} dependsIn(C, C_i)$$

Onde:

$$dependsIn(C, C_i) = \begin{cases} 1, & se\ C_i\ depende\ de\ C \\ 0, & senão \end{cases}$$

A.18.3 Aplicação da fórmula

A Figura 35 é um diagrama com quatro classes. Onde a classe Cliente é uma dependência da classe Venda, a classe Item é uma dependência da classe Venda e a classe Poduto é dependencia da classe Item. Logo:

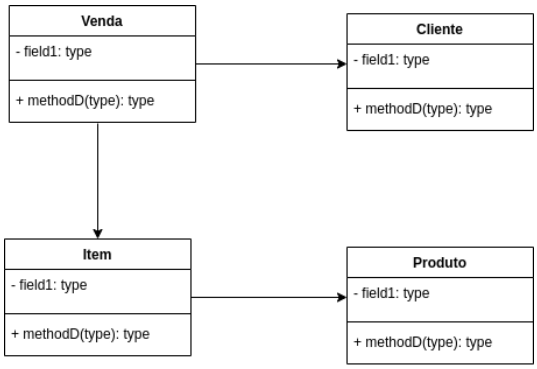


Figura 35 – Diagrama NDepIn (Fonte: Autor)

$$NDepIn(Cliente) = 1$$

$$NDepIn(Venda) = 0$$

$$NDepIn(Item) = 1$$

$$NDepIn(Produto) = 1$$

A.19 Number of Dependencies Out (NDepOut)

A métrica *Number of Dependencies Out* contabiliza o número de dependência que uma determinada classe possui com outras, ou seja, mede o quanto uma classe específica utiliza outras classes (GENERO; PIATTINI; CALERO, 2005).

A.19.1 Fator de qualidade

Essa métrica tem esse aspecto semelhante a métrica anterior, de modo que o fator de qualidade afetado por ela também é o acoplamento, podendo, contudo, apontar algum problema de coesão.

A.19.2 Cálculo

$$NDepOut(C) = \sum_{i=1}^{n} dependsOut(C, C_i)$$

Onde:

$$dependsOut(C, C_i) = \begin{cases} 1, & se\ C\ depende\ C_i \\ 0, & senão \end{cases}$$

A.19.3 Aplicação da fórmula

A Figura 36 é um diagrama com quatro classes. Onde a classe Venda depende das classes Cliente e Item e a classe Item depende da classe Produto. Logo:

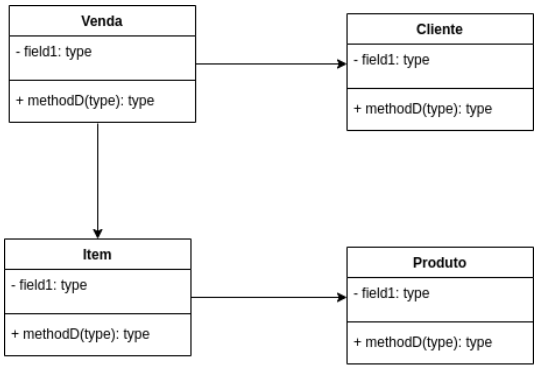


Figura 36 – Diagrama NDepOut (Fonte: Autor)

$$NDepOut(Cliente) = 0$$

$$NDepOut(Venda) = 2$$

$$NDepOut(Item) = 1$$

$$NDepOut(Produto) = 0$$

A.20 Lack of Cohesion of Method - CK (LCOM)

Coesão é uma medida que indica o quão um elemento foca em suas responsabilidades. Em outras palavras, especifica se um elemento, seja ele uma classe, método ou

módulo, tem responsabilidades relacionadas e coerentes. Com isso, um elemento que detém responsabilidades relacionadas e sem alta quantidade de esforço pode ser considerado um elemento com alta coesão (LARMAN, 2012). Uma métrica utilizada para calculo de coesão é *Lack of cohesion of method (LCOM)* que envolve a quantidade de atributos de uma classe e uso destes em outras partes do elemento, em outras palavras, busca a correlação entre as variáveis da classe e a utilização delas nos métodos das classes (HITZ; MONTAZERI, 1995).

A.20.1 Fator de qualidade

A métrica visa ser um indicador da ausência de coesão entre os métodos de uma classe. Dessa forma, quanto maior LCOM, menor será a coesão entre os pares de métodos, haja visto que a ausência de similaridade entre os pares contribui para aumentar o valor medido, enquanto a presença de pares de métodos com similaridade contribuem para diminuir esse valor.

A.20.2 Cálculo

Considere uma classe *C1* com os métodos *{M1, M2, ..., Mn}*. Dado *{Ii}* o conjunto de atributos da classe. Existem n conjuntos desse tipo *{I1}*, ..., *{In}*. Para $P = \{(Ii, Ij) \mid Ii \cap Ij = \emptyset\}$ e $Q = \{(Ii, Ij) \mid Ii \cap Ij \neq \emptyset\}$ se todos n conjuntos *{I1}, ..., In* são $\emptyset$ então $P = \emptyset$ (HITZ; MONTAZERI, 1995).

$$LCOM = \begin{cases} |P| - |Q|, & se |P| > |Q| \\ 0, & senão \end{cases}$$

A.20.3 Aplicação da fórmula

Dado uma classe C com três métodos *{x, y, z}* e com 6 atributos *{A, B, C, D, E, F}*. Onde o método x acessa os atributos *{a, b, c}*, o método y acessa os atributos *{d, e}* e o método y acessa os atributos *{e, f}* é possível dizer que

$$P = \{< x, y >, < x, z >\}$$

$$Q = \{< y, z >\}$$

$$LCOM = 2 - 1$$

$$LCOM = 1$$

```
class C:
def __init__(self):
    self.A = None
    self.B = None
    self.C = None
    self.D = None
    self.E = None
    self.F = None

def X(self):
    # Use self.A, self.B e self.C
    ...

def Y(self):
    # Use self.D e self.E
    ...

def Z(self):
    # Use self.E e self.F
    ...
```

Listing 6 – Código fonte - LCOM

A.21 Lack of Cohesion Metrics - HS (LCOM)

A métrica *Lack of Cohesion Metrics - HS* calcula a falta de coesão de um tipo através dos métodos da classe. Os tipos mais bem projetados terão métodos que acessam os mesmos campos. Caso contrário irão acessar tipos de dados muito diferentes da classe, isso é uma indicação de que a classe pode ter várias responsabilidades (HENDERSON-SELLERS, 1995).

A.21.1 Fator de qualidade

Quando essa métrica apresenta um valor alto, é um indicativo que a coesão geralmente da classe deve ser revista, pois indica que uma classe tem múltiplas responsabilidades distintas. A solução é identificar essas responsabilidades e extrair os métodos e dados associados em classes separadas.

A.21.2 Cálculo

$$LCOM = \frac{p - m}{1 - m}$$

Onde:

```java
public class Pessoa{
    double peso;
    double altura;
    double IMC;
    public double altura(){
        return altura;
    }
    public double peso(){
        return peso;
    }
    public double IMC(){
        IMC = peso/altura * altura;
        return IMC;
    }
}
```

Listing 7 – Código fonte - LCOM

- p é o número médio de métodos que acessam cada campo
- m o número de métodos que possuem o mesmo tipo

A.21.3 Aplicação da fórmula

Considere a Listing 7 para exemplo de cálculo da métrica.

$$m = 3$$
$$p = \frac{(1+1+2)}{3} = 1.33$$
$$LCOM = \frac{1.33-3}{1-3} = \frac{-1.67}{-2} = 0.83$$

A.22 Lack of Cohesion Metrics - Pairwise Field Irrelation (LCOM)

A coesão é um conceito importante na programação OO. Indica se uma classe representa uma única abstração ou várias abstrações. A ideia é que, se uma classe representar mais de uma abstração, ela deve ser refatorada em mais de uma classe, cada uma representando uma única abstração. A métrica *Lack of Cohesion Metrics - Pairwise Field Irrelation* mede coesão de uma classe. É através desta métrica que é possível verificar se uma classe está ou não bem definida (IBRAHIM; KAMEL; HASSAN, 2016).

A.22.1 Fator de qualidade

É uma métrica fundamental que aponta o quanto uma classe está mal formulada e misturam diferentes atributos e conceitos. Quando essa métrica aponta valores altos é necessário uma revisão e refatoração na classe, pois é um indicativo que a classe não está bem definida.

A.22.2 Cálculo

Para o cálculo de coesão a seguir, será utilizado a definição de falta de coesão de Pairwise Field Irrelation. Onde:

- M é o conjunto de métodos definidos pela classe
- F é o conjunto de atributos definidos pela classe
- Mf é o subconjunto M dos métodos que acessam o campo f, onde f é um membro de F

Então a coesão de Pairwise Field Irrelation é a distância média de Jaccard entre Mf1 e Mf2, onde f1 $\neq$ f2.

A média de Jaccard pode ser calculada por:

$$d_j(A, B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$$

A.22.3 Aplicação da fórmula

Considere a Listing 8 para exemplo de cálculo da métrica.

$$M = \{calculaAltura, calculaPeso, calculaIMC\}$$
$$Mf1 = \{calculaAltura, calculaPeso, calculaIMC\}$$
$$Mf2 = \{calculaIMC\}$$

$$|Mf1 \cup Mf2| = 3$$
$$|Mf1 \cap Mf2| = 1$$

$$d_j(Mf1, Mf2) = \frac{3-1}{3} = \frac{2}{3} = 0.66$$

```
1  public class Pessoa{
2      double peso;
3      double altura;
4      double IMC;
5      public double altura(){
6          return altura;
7      }
8      public double peso(){
9          return peso;
10     }
11     public double IMC(){
12         IMC = peso/altura * altura;
13         return IMC;
14     }
15 }
```

Listing 8 – Código fonte - LCOM

A.23 Cohesion Among Methods of Class (CAMC)

A métrica *Cohesion Among Methods of Class* calcula a relação entre métodos de uma classe com base na lista de parâmetros dos métodos (GENERO; PIATTINI; CALERO, 2005). Portanto, é utilizado os tipos dos parâmetros que o método usa como entrada para o cálculo da coesão.

A.23.1 Fator de qualidade

Quanto maior o valor do CAMC pior é a atribuição das responsabilidades da classe, ou seja, menor é a coesão. Isso implica, geralmente, em classes grandes e muito complexas, sendo difícil manter e modificar a classe.

A.23.2 Cálculo

$$CAMC(C) = \frac{\sum_{i=1}^{n} P_i}{T} \times N$$

Onde:

- Pi representa a quantidade de tipos de parâmetros
- N representa o número de métodos
- T é o número de parâmetros distintos da classe

A.23.3 Aplicação da fórmula

A Figura 37 representa um sistema com uma classe. Onde existem três métodos. O methodA tem dois parâmetros e os métodos methodB e methodC tem um parâmetro cada. Logo:

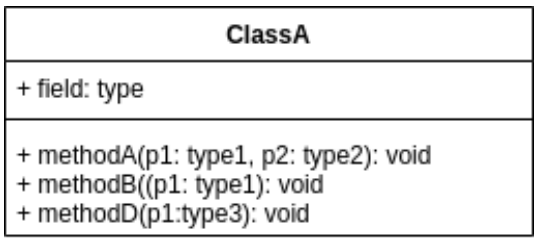


Figura 37 – Diagrama CAMC (Fonte: Autor)

$$CAMC(ClassA) = \frac{3}{3 \times 3}$$
$$CAMC(ClassA) = 0.33$$

A.24 Number of Children (NOC)

A métrica *Number of Children* tem o objetivo de medir a quantidade de filhos em primeiro nível de uma classe, ou seja, conta em números absolutos a quantidade de subclasses imediatamente subordinadas a uma outra classe (GENERO; PIATTINI; CALERO, 2005).

A.24.1 Fator de qualidade

Quanto mais filhos diretos uma classe tem, mais classes isso pode acabar afetando por conta da herança, tornando assim mais difícil realizar uma manutenção.

A.24.2 Cálculo

$$NOC(C) = \sum_{i=1}^{n} isChild(C, C_i)$$

Onde:

$$isChild(C, C_i) = \begin{cases} 1, & se\ C_i\ herda\ diretamente\ de\ C \\ 0, & senão \end{cases}$$

A.24.3 Aplicação da fórmula

A Figura 38 é um diagrama com cinco classes. Onde a classe Room tem dois filhos e a classe DecoratorRoom também tem dois filhos, as demais classes não tem filhos. Logo:

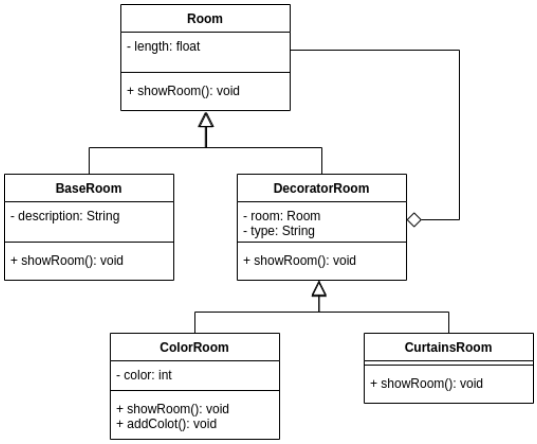


Figura 38 – Diagrama NOC (Fonte: Autor)

$$NOC(Room) = 2$$

$$NOC(ColorRoom) = 0$$

$$NOC(CurtainsRoom) = 0$$

$$NOC(BaseRoom) = 0$$

$$NOC(DecoratorRoom) = 2$$

A.25 Number of Methods Overridden (NMO)

A métrica *Number of Methods Overridden* calcula o número total de métodos sobreescritos por uma classe. Isto é, dada uma hierarquia, é contabilizado quantos métodos de uma determinada classe são reescritos (GENERO; PIATTINI; CALERO, 2005).

A.25.1 Fator de qualidade

O número de operações redefinidas desempenha um papel na especialização da classe e deve ser mantido em uma proporção que justifique a herança. Muitas operações de redefinição de métodos da classe pai implicam em uma diferença muito grande com a classe pai e a herança passa a fazer menos sentido.

A.25.2 Cálculo

$$NMO(C) = \sum_{i=1}^{n} isOverride(M_i)$$

Onde:

$$isOverride(M_i) = \begin{cases} 1, & se\ M_i\ for\ um\ método\ sobreescrito \\ 0, & senão \end{cases}$$

A.25.3 Aplicação da fórmula

A Figura 39 é um diagrama com quatro classes. Onde a ClasseD sobreescreve dois métodos da ClasseA e a ClasseC sobreescreve um da ClasseA. Logo:

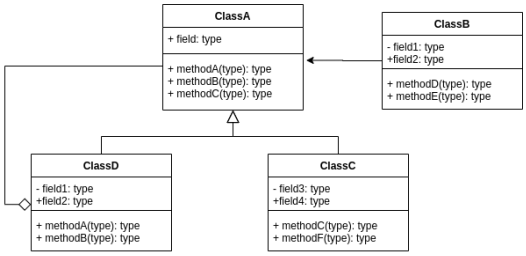


Figura 39 – Diagrama NMO (Fonte: Autor)

$$NMO(ClasseA) = 0$$

$$NMO(ClasseB) = 0$$

$$NMO(ClasseC) = 2$$

$$NMO(ClasseD) = 1$$

A.26 Specialization Index (SIX)

A métrica *Specialization Index* visa avaliar o quanto determinada classe sobrescreve o comportamento de suas superclasses. Quanto maior o valor do índice de especialização, maior é a execução de comportamentos especializados em tempo de execução, indicando uma classe mais complexa, devendo receber maior esforço de testes.

A.26.1 Fator de qualidade

Esta métrica visa avaliar o quanto determinada classe sobrescreve o comportamento de suas superclasses. De acordo com os autores da métrica, quanto maior o valor do índice de especialização, maior é a execução de comportamentos especializados em tempo de execução, indicando uma classe mais complexa, devendo receber maior esforço de testes.

A.26.2 Cálculo

$$SIX = \sum_{i=1}^{n} isOverride(M_i) \times \frac{DIT}{NOM}$$

Onde:

$$isOverride(M_i) = \begin{cases} 1, & se\ M_i\ for\ um\ método\ sobreescrito \\ 0, & senão \end{cases}$$

- Mi: representa os métodos da classe
- DIT: representa a profundidade na hierarquia
- NOM: os métodos da classe

A.26.3 Aplicação da fórmula

A Figura 40 é um diagrama com cinco classes. Onde a classe Room não sobreescreve nenhum método e as demais sobreescrevem um método cada. Logo:

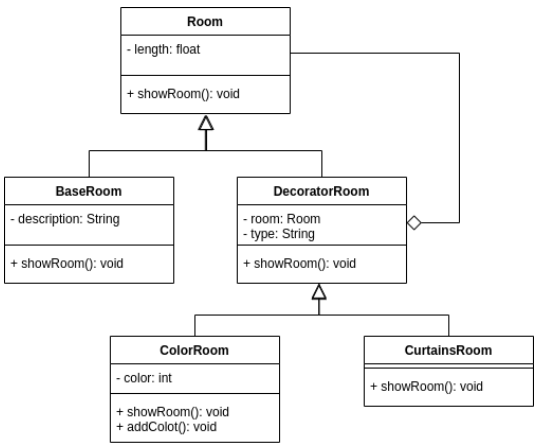


Figura 40 – Diagrama SIX (Fonte: Autor)

$$SIX(Room) = 0$$

$$SIX(ColorRoom) = 1 \times \frac{2}{2} = 1$$

$$SIX(CurtainsRoom) = 1 \times \frac{2}{1} = 2$$

$$SIX(BaseRoom) = 1 \times \frac{1}{1} = 1$$

$$SIX(DecoratorRoom) = 1 \times \frac{1}{1} = 1$$

A.27 Method Inheritance Factor (MIF)

A métrica *Method Inheritance Factor* mede a um fator relacionado a quantidade de métodos herdados dos pais, isto implica em uma complexidade proporcional a quantidade de métodos herdados (AL-JA'AFER; SABRI, 2004).

A.27.1 Fator de qualidade

A herança - que é considerada um dos principais conceitos de programação orientada a objetos, evitando duplicação e reduzindo a redundância. - afeta diversos fatores de qualidade, como: reúso, manutenibilidade, extensibilidade, ocorrência de erros, flexibilidade e entendimento.

A.27.2 Cálculo

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

Onde:

- Mi(Ci): métodos herdados pela classe Ci
- Ma(Ci) = Md(Ci) + Mi(Ci): métodos da classe Ci
- Md(Ci): métodos definidos na classe Ci
- TC: Total de classes

A.27.3 Aplicação da fórmula

A Figura 41 apresenta um diagrama de classes com duas classes, a ClasseA tem três métodos e a ClasseB tem um método, além disso a ClasseB herda os três métodos da ClasseA. Logo:

$$AIF = \frac{\sum(0,3)}{\sum(3,1)}$$
$$AIF = \frac{4}{4}$$
$$AIF = 0.75$$

A.28 Attribute Inheritance Factor (AIF)

A métrica *Attribute Inheritance Factor* mede um fator relacionado a quantidade de atributos herdados (AL-JA'AFER; SABRI, 2004).

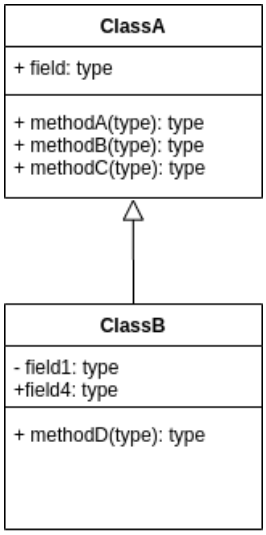


Figura 41 – Diagrama MIF (Fonte: Autor)

A.28.1 Fator de qualidade

A Herança - que é considerada um dos principais conceitos de programação orientada a objetos, evitando duplicação e reduzindo a redundância. - afeta diversos fatores de qualidade, como: reúso, manutenibilidade, extensibilidade, ocorrência de erros, flexibilidade e entendimento.

A.28.2 Cálculo

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

Onde:

- Ai(Ci): atributos herdados pela classe Ci
- Aa(Ci) = Ad(Ci) + Ai(Ci): atributos da classe Ci
- Ad(Ci): atributos definidos na classe Ci
- TC: Total de classes

A.28.3 Aplicação da fórmula

A Figura 42 apresenta um diagrama de classes com duas classes, a ClasseA tem um atributo e a ClasseB tem dois atributos, além disso a ClasseB herda um atributo da ClasseA. Logo:

$$AIF = \frac{\sum(0,1)}{\sum(1,2)}$$

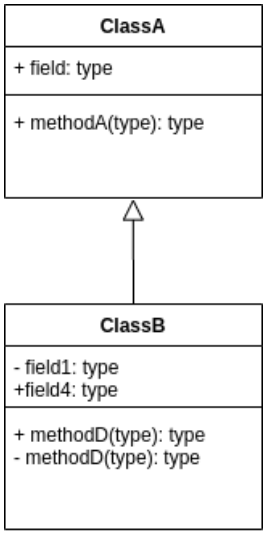


Figura 42 – Diagrama AIF (Fonte: Autor)

$$AIF = \frac{1}{3}$$
$$AIF = 0.33$$

A.29 Polymorphism Factor (POF)

A métrica *Polymorphism Factor* representa o número real de possíveis situações polimórficas diferentes (AL-JA'AFER; SABRI, 2004). Portanto, a métrica POF mede quantitativamente a utilização do polimorfismo no sistema.

A.29.1 Fator de qualidade

O polimorfismo surge da herança. Abreu afirma que, em alguns casos, Os métodos predominantes reduzem a complexidade, aumentando assim a manutenção.

A.29.2 Cálculo

$$POF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) \times DC(C_i)]}$$

Onde:

- Mo(Ci): métodos sobrescritos na classe Ci
- Mn(Ci) novos métodos na classe Ci
- DC(Ci): número de decendentes da classe Ci
- TC: Total de classes

A.29.3 Aplicação da fórmula

A Figura 43 apresenta um diagrama de classes com três classes, a ClasseA tem três métodos, a ClasseB tem três métodos, sendo que dois são sobreescritos, e a ClasseC com um método. Logo:

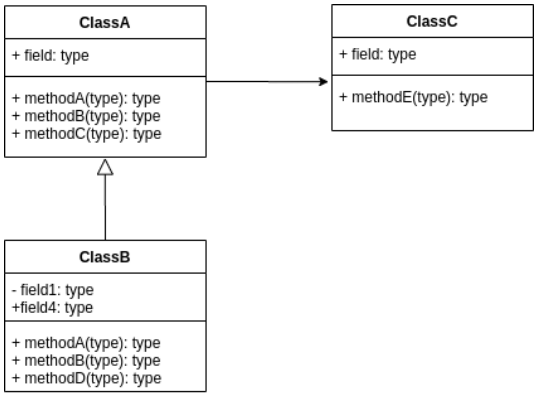


Figura 43 – Diagrama POF (Fonte: Autor)

$$MHF = \frac{\sum(0,2,0)}{\sum(3*1,1*0,1*0)}$$
$$MHF = \frac{2}{3}$$
$$MHF = 0.66$$

A.30 Method Hiding Factor (MHF)

A métrica *Method Hiding Factor* mede um fator referente a abstração das interfaces das classes, onde as os métodos públicos devem ser apenas uma forma de se acessar uma funcionalidade, onde tal é responsável por compor a funcionalidade por completo (AL-JA'AFER; SABRI, 2004).

A.30.1 Fator de qualidade

MHF baixo indica uma dificuldade de abastração, com muitos erros possíveis. Um MHF alto indica que poucas funcionalidades são implementadas. Acredita-se que algo em torno do intervalo definido entre 8% e 25% seja o ideal (AL-JA'AFER; SABRI, 2004).

A.30.2 Cálculo

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}$$

Onde:

- Mh(Ci): métodos ocultos na classe Ci
- Md(Ci) = Mv(Ci) + Mh(Ci): métodos definidos na classe Ci
- Mv(Ci): métodos visíveis na classe Ci
- TC: Total de classes

A.30.3 Aplicação da fórmula

A Figura 44 apresenta um diagrama de classes com duas classes, a ClasseA tem um três métodos, dois privados e um público, e a ClasseB tem quatro métodos, sendo três privados e um público. Logo:

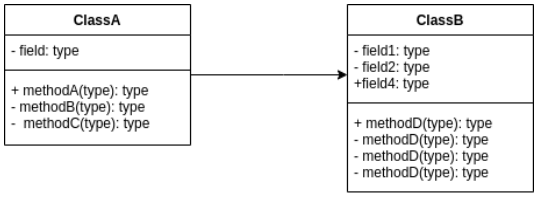


Figura 44 – Diagrama POF (Fonte: Autor)

$$MHF = \frac{\sum(2,3)}{\sum(3,4)}$$
$$MHF = \frac{5}{7}$$
$$MHF = 0.71$$

A.31 Attribute Hiding Factor (AHF)

A métrica *Attribute Hiding Factor* tem o objetivo de medir o uso das informações ocultas no sistema, isto posto, a métrica AHF utiliza do conceito de encapsulamento para lidar com a complexidade do sistema (AL-JA'AFER; SABRI, 2004), de forma ideal todos os atributos deveriam ser ocultos e acessados e manipulados apenas pelas classes que as declaram.

A.31.1 Fator de qualidade

Um valor alto de AHF tende a indicar uma falta de modularidade do sistema, onde partes do sistema são acopladas, isto é, existe uma dependência uma das outras. Desta forma, os fatores de reúso, manutenção e segurança são evidenciados por esta métrica.

A.31.2 Cálculo

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}$$

Onde:

- Ah(Ci): atributos ocultos na classe Ci
- Ad(Ci) = Av(Ci) + Ah(Ci): atributos definidos na classe Ci
- Av(Ci): atributos visíveis na classe Ci
- TC: Total de classes

A.31.3 Aplicação da fórmula

A Figura 45 apresenta um diagrama de classes com duas classes, a ClasseA tem um atributo privado e a ClasseB tem três atributos, sendo dois privados e um público. Logo:

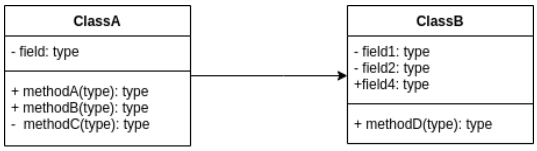


Figura 45 – Diagrama AHF (Fonte: Autor)

$$AHF = \frac{\sum(1,2)}{\sum(1,3)}$$
$$AHF = \frac{3}{4}$$
$$AHF = 0.75$$

A.32 Duplicação de código

A duplicação de código envolve no mínimo um par de fragmentos de código onde um é a cópia do outro (RIEGER; DUCASSE; LANZA, 2004). As métricas relacionadas a dúplicação de código tentam verificar a repetição de código, tal característica deixa o código mais difícil de se manter, uma vez que se for necessário alguma mudança é preciso alterar em todas as réplicas.

```
def a(self, value):
    rest  = value % 100
    result = rest / self.total
    return result

def b(self, value):
    rest  = value % 100
    result = rest / self.total
    if result > 1:
        result = result * 0.75
    return result
```

Listing 9 – Código fonte - Duplicação de código

A.32.1 Fator de qualidade

Caso um trecho de código duplicado necessite ser modificado, deve ser mudado em todas as ocorrências, e por isso, a duplicação tende a dificultar a manutenção e modificações posteriores ao desenvolvimento.

A.32.2 Cálculo

A forma de cálculo pode variar, mas no geral é utilizado o *SLOC* para comparar o tamanho do código em comum entre dois trechos de código fonte. Isso pode ser dado como número absoluto ou porcentagem ao ser comparado entre dois métodos, duas classes ou dois arquivos.

A.32.3 Aplicação da fórmula

Dado os dois métodos na Listing 9 é vísivel que a duplicação está presente no cálculo do valor de retorno, é possível dizer que existem duas linhas ininterruptas de replicação entre os métodos a e b.