UM ESTUDO SOBRE A REDUÇÃO DO ALTO ACOPLAMENTO E BAIXA COESÃO NO DESIGN DE CLASSES EM JAVA UTILIZANDO OS PRINCÍPIOS SOLID

2016

UM ESTUDO SOBRE A REDUÇÃO DO ALTO ACOPLAMENTO E BAIXA COESÃO NO DESIGN DE CLASSES EM JAVA UTILIZANDO OS PRINCÍPIOS SOLID

Autor: Yuri Ramalho Pinheiro[1]

Orientador: Eder Oliveira[2]

Curso: MBA em Projetos de Aplicação Java

RESUMO

O objetivo deste artigo é propor os princípios SOLID como alternativa para reduzir o alto acoplamento e a baixa coesão no design de classes em Java. Através de pesquisa bibliográfica, digital e estudo de caso foi elaborado o assunto ao qual este artigo irá tratar. Os resultados a serem apresentados serão descritos através de diagramas de classes e códigos. Após o estudo deste artigo será possível utilizar os princípios SOLID como técnica de ajuda aos desenvolvedores de software que pretendam evitar os problemas que estão relacionados ao gerenciamento do acoplamento e da coesão e contribuir com o desenvolvimento de qualidade em Java.

Palavras chaves: alto acoplamento, baixa coesão, SOLID, Java, software.

ABSTRACT

This article's aim is to offer SOLID principles as an alternative to reduce high coupling and low cohesion in java class design. This article was based on both online and offline bibliographical research and case study. The results here presented are described by means of class diagrams and codes. The analysis revealed positive results, such as coupling reduction and increased cohesion after SOLID principles were applied in classes. By studying this article it will be possible to employ SOLID principles as a help technique to software developers who intend on avoiding problems related to coupling and cohesion management and pursuing a high quality development in Java.

Key words: high coupling, low cohesion, SOLID, Java and software.

---

1 - Yuri Ramalho Pinheiro é profissionalizado em Técnico em Informática pelo SENAI e graduado em Análise e Desenvolvimento de Sistemas pela UNOPAR (Universidade Norte do Paraná).

2 - Eder Oliveira é professor e orientador na Instituição de Ensino IGTI e especialista em tópicos avançados de Tecnologia da Informação.

1 - INTRODUÇÃO

O desenvolvimento de software utilizando o paradigma da Orientação a Objetos é uma forma avançada de abstrair problemas do mundo real para o mundo da programação. Este paradigma é utilizado por muitas linguagens de programação, inclusive Java. Mas ele pode ser um problema caso não seja aplicado da maneira correta.

Quando se propõe desenvolver em Java, mas não se utiliza da maneira correta para planejar e desenvolver o design do código é provável que ocorrera dois problemas básicos na organização das classes: alto acoplamento e baixa coesão**.** Porem mesmo sendo considerados básicos em um projeto, corrigi-los pode ser um trabalho complexo**.**

Este estudo busca estudar recurso que possam ser utilizados para evitar que estes problemas comprometam o desenvolvimento e a qualidade do software. Então a dúvida a qual buscamos esclarecer é a seguinte: como é possível reduzir o alto acoplamento e baixa coesão no design de classes em Java?

Ao tentar identificar soluções para estes problemas é possível encontrar muitas formas diferentes de resolução que são descritas em vários artigos e livros que tratam de assuntos como Padrões de Projetos de Software, Arquitetura de software, Programação Orientada a Objetos entre outros desde os anos 60 até momento atual, mas esta pesquisa buscou abordar conhecimentos do cenário mais atual.

Uma possibilidade encontrada foi os princípios SOLID, um conjunto de princípios utilizados para se obter uma melhor qualidade do software através dos recursos da orientação a objetos e tratar de problemas relacionados ao acoplamento e coesão. Este artigo tem como objetivo propor os princípios SOLID como alternativa para reduzir o alto acoplamento e a baixa coesão do design de classes em Java.

Este artigo se divide em sessões, a próxima a ser apresentada e sobre a Metodologia ao qual se aplicou este estudo. Em seguida, será apresentado a sessão de Referencial Teórico para expor os conceitos sobre o tema pesquisado. Consequentemente, será apresentado a sessão de Resultados para descrever o impacto da utilização dos princípios SOLID sobre os problemas relacionados ao acoplamento e coesão. E por fim a Conclusão para propor algumas sugestões sobre o tema.

2 - METODOLOGIA

A pesquisa realizada neste artigo pode ser classificada como **exploratória**. Isto porque deve a pesquisa em mãos analisar a abrangência da área ao qual foi escolhido este tema e sua dimensão para entender mais sobre o assunto, buscando referências bibliográficas e digitais através livros e da ferramenta de pesquisa do Google, para melhor descrever e explicar dado fenômeno.

Utilizou-se a **abordagem qualitativa**, pois buscamos o motivo pelo qual o alto acoplamento e a baixa coesão se tornaram um grande problema para o desenvolvimento de software.

Sua natureza é do tipo **aplicada**, onde foi feito uma pesquisa para descrever como os princípios SOLID, identificados, podem ajudar a evitar o problema proposto e em seguida será apresentado alguns exemplos de utilização para melhor entendimento.

Quanto à metodologia o trabalho em mãos, foi escolhido o **método de estudo de caso**. Esta opção se justifica porque o método escolhido permite investigar o problema proposto em várias fontes de informação e em seguida propor a análise dos dados coletados através de testes.

Sobre o levantamento de dados sobre o assunto, inicialmente buscou-se fazer uma ampla pesquisa em livros e artigos digitais, sobre o tema proposto, autores e outros artigos científicos que abordam o mesmo tema para identificar lacunas que ainda precisassem ser abordadas e complementadas.

Sobre a verificação da qualidade da fonte fornecedora dos dados, também foi de suma importância verificar a veracidade das informações coletadas. Foi preferido os dados aos quais as fontes bibliográficas fazem referência a autores com um maior grau de graduação como mestres e doutores ou com experiência comprovada sobre o assunto na descrição de livros.

A partir da filtragem dos dados selecionados utilizou-se da seguinte abordagem para descrever o artigo:

- Foi feita uma leitura dos livros: Introdução a Arquitetura e Design de Software: Uma visão sobre a Plataforma Java; Orientação a Objetos e SOLID para ninjas: Projetando classes flexíveis. Selecionados para entender com mais detalhes o assunto abordado.

- Utilizou-se da plataforma de pesquisa ResearchGate para identificar artigos que estivessem relacionados ao tema.

- Buscou fatos, como citações, abordados pelos autores dos livros e artigos escolhidos para melhor enfatizar a importância o problema e a solução a ser descrita por esta pesquisa.

- Através do entendimento do assunto e dos fatos abordados, criou-se exemplos de diagramas de classes utilizando a Linguagem de Modelagem Unificada – UML (Unified Modeling Language) que é uma forma de descrever modelos de diagramas para elaboração construção de projetos de softwares.

- Também foi necessário a criação de algumas linhas de código utilizando a plataforma de desenvolvimento de software chamada Eclipse, para auxiliar no entendimento dos diagramas criados.

- Por fim, foi descrito na sessão de Resultados deste artigo a análise feita sobre a aplicação dos princípios SOLID nos conceitos de acoplamento e coesão.

3 - REFERENCIAL TEÓRICO

3.1 - Acoplamento

Acoplamento “é o quanto dois elementos estão amarrados entre si e quanto as alterações no comportamento de um afetam o de outro” (SILVEIRA; SILVAIRA; LOPES; MOREIRA; STTEPAT; KUNG, 2015, p. 76).

Segundo Gilliard Cordeiro (2013) que trata do tema acoplamento em seu livro de (CDI – Integre as dependências e contextos do seu código Java) e faz referência a este termo, como a medida da sociabilidade de um objeto, ou seja, o quanto ele se relaciona com outros objetos e a força entre o relacionamento.

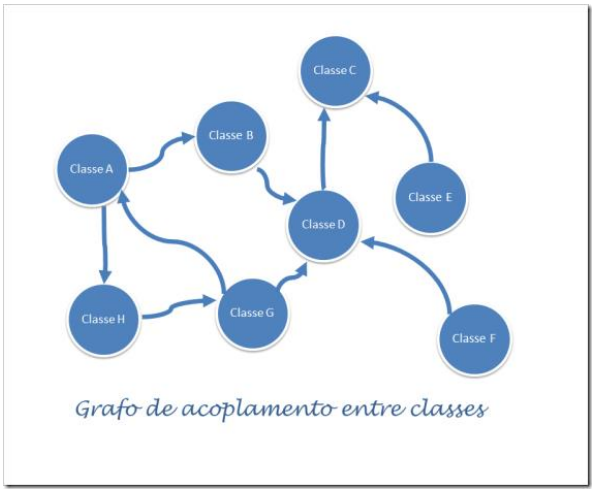

Figura I - Acoplamento entre classes

Fonte: Leandro Daniel, 2012

3.1.1 - Alto Acoplamento

Quando uma classe se relaciona com muitas outras classes e sofre influência em seu comportamento, a partir de alterações ocorridas em suas classes de dependência, o relacionamento entre elas passa a ser considerado um alto acoplamento.

Um dos problemas do alto acoplamento entre classes é que se uma mudança ocorrer em uma delas, ele será propagado para as outras classes de dependência, podendo impactar em dificuldade de manutenção do código fonte, fazendo com que o programador precise verificar todos os pontos do código onde aquela classe foi utilizada para corrigir os erros.

Outro problema do alto acoplamento segundo Mauricio Aniche (2012) é que ele dificulta a reutilização das classes, pois caso seja necessário utilizar um conjunto de classes em outro projeto por exemplo, será necessário levar as classes de dependência também, ocasionando problemas de complexidade no código fonte.

3.1.2 - Acoplamento eferente e aferente (conceitos)

Quando uma classe depende de outras classes, maior é o acoplamento entre elas e mais frágil ela se torna, então considera-se que este é um acoplamento eferente. (ANICHE, 2015)

Já o acoplamento aferente mede quantas classes dependem da classe principal, este é o tipo de acoplamento que devemos nos preocupar quando queremos ver se a classe está estável, pois uma mudança neste tipo de relacionamento pode afetar todo o sistema. (ANICHE, 2015)

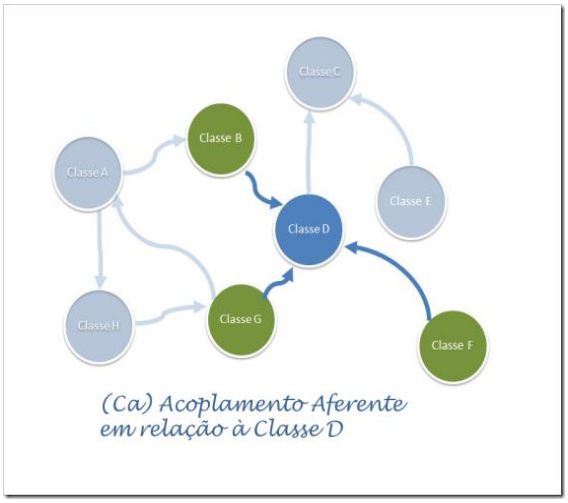

Figura II - Acoplamento aferente

Fonte: Leandro Daniel, 2012

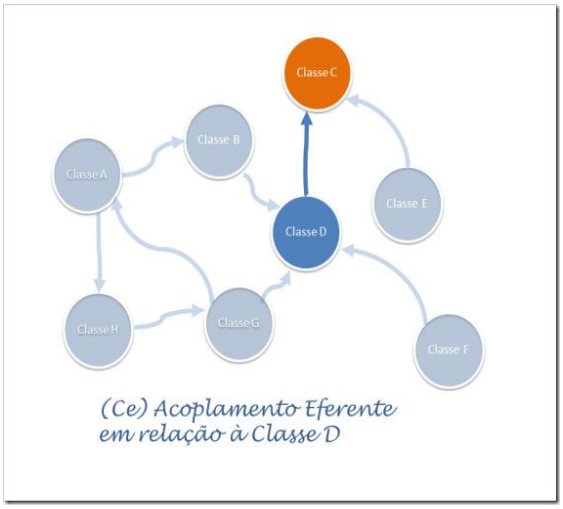

Figura IIII - Acoplamento eferente

Fonte: Leandro Daniel, 2012

Quando estamos criando código em Java é inevitável criar acoplamento entre classes, pois precisamos criar relacionamentos entre as classes para podermos usufruir dos recursos da Orientação a Objetos. Cabe a nós tentar identificar se os relacionamentos que estamos criando são favoráveis ou desfavoráveis para o nosso código, sempre buscando por classes estáveis, ou seja, que sofrem poucas mudanças em suas implementações.

3.2 - Coesão

O autor Mauricio Aniche trata do tema coesão em seu livro (Test Driven Development: Teste e Design no mundo real) e faz referência a este termo como “Uma classe coesa é justamente aquela que possui apenas uma única responsabilidade. Em sistemas orientados a objetos, a ideia e sempre buscar por classes coesas”. (ANICHE, 2012, p. 73)

Os autores, Kathy Sierra e Bert Bates, do livro (Sun Certified Programmer for Java 6), destacam a coesão como o grau de objetividade de uma classe, sendo assim quanto mais focalizada está uma classe em realizar seu objetivo mais coesa ela estará, em contrapartida, quanto menor for o foco da classe em realizar sua responsabilidade, menos coesa ela será. (2008)

2.1 - Baixa Coesão

“Uma classe altamente coesa tem responsabilidades e propósitos claros e bem definidos, enquanto uma classe com baixa coesão tem muitas responsabilidades diferentes e pouco relacionadas”. (LUQUE, 2010, p. 3).

Divergent Change é um nome atribuído para uma classe: “quando a classe não é coesa, e sofre alterações constantes, devido as suas diversas responsabilidades” (ANICHE, 2015, p. 123).

Veja abaixo o diagrama da classe (NotaFiscal) e sua baixa coesão. Ela c deve possuir a responsabilidade de descriminar serviços, mas perceba que foi projetada para fazer mais que sua obrigação, então ela também é responsável por salvar dados no banco de dados, enviar e-mail, cadastrar cliente, ou seja não está coesa. Caso seja necessário

fazer alterações nas regras de negócio desta classe, ela pode continuar aumentado e ficando mais complexa.

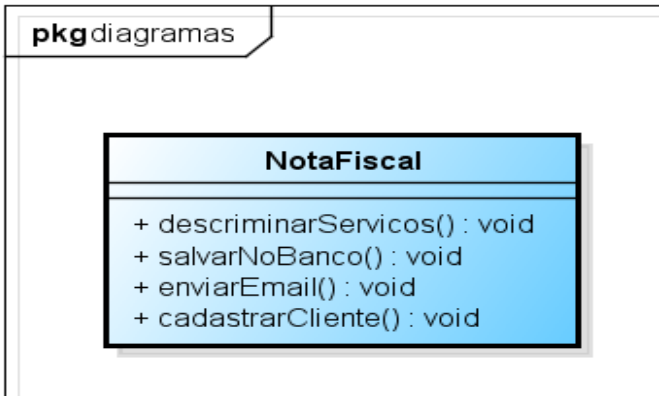

Figura IV – Modelo de diagrama da classe Nota Fiscal

Fonte: Yuri Ramalho Pinheiro, 2016

3 - Princípios SOLID

Os princípios SOLID são o acrônimo da junção de 5 letras que representam os princípios propostos por Martin C. Robert também conhecido como "Uncle Bob", em seu artigo sobre Principles of Object Oriented Design (Princípios de projetos Orientados a Objetos). Eles não foram criados por Martin, mas identificados por ele e reunidos com o propósito de gerenciar dependências de códigos com difíceis alterações, frágeis e não reutilizáveis. Seu objetivo era atingir códigos mais flexíveis, robustos e reutilizáveis para melhor a qualidade dos softwares. Ele também incluiu estes princípios no contexto do design de classes da orientação a objetos.

Figura V - Princípios SOLID

Fonte: Eduardo Pires - Treinamentos e consultorias, 2015

3.1 – SRP - Single Responsability Principle (Princípio da Responsabilidade Única)

Segundo Silva apud Martin (2013) este princípio tem como base a coesão. Classes com maior coesão são mais fáceis de entender e manter. Ele destaca que uma classe dever ter somente uma razão para mudar. Caso exista mais de uma razão para mudar, significa que as responsabilidades responsáveis por causar as mudanças devem ser separadas em outras classes. Sempre que os requisitos de uma classe mudam, suas responsabilidades também podem sofrem alterações, isso pode prejudicar o funcionamento da classe, tornando ela frágil e proporcionando problemas no projeto.

Ele conclui afirmando que este princípio é o mais simples, mas um dos mais difíceis de aplicar corretamente, porque a definição de responsabilidades pode ser interpretada de pontos de vistas diferentes, um mesmo modelo de classe pode ser criado de maneira diferente por equipes de desenvolvimento distintas.

Segundo Aniche (2015) em seu livro de Orientação a Objetos e SOLID para Ninjas, foi utilizado este princípio com o objetivo de gerenciar a coesão. Ele destacou que classes coesas são menores, simples, fáceis de reutilizar e propagam menos problemas para outras classes. Para ele é mais difícil enxergar as responsabilidades das classes logo no início do projeto, por isso colocamos mais código do que o necessário. Ele propõe que para encontrar classes que não são coesas devemos buscar por: classes que possuem muito métodos, sofrem muitas modificações e que nunca param de crescer.

3.2 – (OCP) - Open Closed Principle (Principio do Aberto/Fechado)

Segundo Silva (2013) este princípio surgiu do trabalho de Bertrand Meyer, reconhecido como autoridade no paradigma de Orientação a Objetos. Ele propõe evitar alterações em classes já estáveis e caso seja necessário faze-las, devemos estender os novos comportamentos para outras classes, sem a necessidade de modificar um conjunto de classe pré-existentes. Este princípio pode ser aplicado em módulos (classes, entidades e funções) de um sistema.

- Módulos devem ser abertos para extensão: isso significa que o comportamento do módulo “pode ser estendido, ou seja, pode-se fazer o módulo adotar comportamentos novos ou diferentes com as mudanças nos requisitos, ou para atender novas necessidades da aplicação”. (SILVA, 2013, p. 60)

- Módulos devem ser fechados para modificações: “o código-fonte de um dado módulo é inviolável. Ninguém está autorizado a alterar o código desse módulo”. (SILVA, 2013, p. 60)

Para manter módulos abertos para extensão e fechados para modificação Silva (2013) destaque que o princípio sugere a utilização da Abstração, um dos recursos da orientação a objetos. Através dela será possível criar classes abstratas como base e caso seja necessário implementar novos comportamentos, devemos utilizar a Herança para estender as características desta classe outras classes concretas.

Segundo Aniche (2015) em seu livro de Orientação a Objetos e SOLID para Ninjas, foi utilizado este princípio com o objetivo de gerenciar a coesão e acoplamento. Ele destacou este princípio como uma solução para evitarmos a utilização de expressão de condição como (if/else) que pode deixar o código muito complexo à medida que novas condições vão sendo adicionadas na expressão. Mas o que isso tem a ver com acoplamento e coesão? Quando utilizamos este tipo de expressão em nossas classes corremos os seguintes riscos:

- A classe irá continuar crescendo e se tornado instável, ou seja, estará aberta para modificações.

- A expressão (if/else) é uma forma de programação estruturada, ou seja, o inverso da programação Orientada a Objetos.

- O código ficará complexo para entendimento e manutenção.

3.3 (LSP) - Liskov Substitution Principle (Principio da Substituição de Liskov)

Segundo Robson Castilho (2016), este princípio foi criado por Barbara Liskov em 1988 e foi introduzido por Robert C. Martin como um dos princípios SOLID. Este princípio está muito relacionado com o principio OCP e a utilização de Herança e Polimorfismo, então ao aplicar o OCP de forma correta você garante a aplicação do LSP.

Segundo ele a definição mais usada para este princípio é: classes derivadas devem poder ser substituídas por suas classes base, ou seja, deve ser possível utilizar os comportamentos das classes filhas (classe derivadas) através da classe pai (classe base). Este princípio está ligado ao acoplamento de abstrações e fortemente ligado a pré-condições e pós-condições, que ligam o LSP ao Projeto por Contrato.

> Uma pré-condição é um contrato que precisa ser satisfeito antes de um método ser invocado. Uma pós-condição, em contrapartida, deve ser verdadeira após a conclusão da execução do método. Se uma pré-condição não é conhecida, o método não deve ser invocado, e se uma pós-condição não é conhecida, o método não retorna. (SILVA, 2013, p. 63)

3.4 – (ISP) - Interface Segregation Principle (Princípio da Segregação de Interface)

Segundo Silva (2013), assim como o Single Responsability Principle é aplicado em classes, este princípio tem o objetivo de criar Interfaces muitos coesas, ou seja, as classes que a implementarem devem assumir apenas uma única responsabilidade e evitar múltiplos papéis. Quando classes são forçadas a depender de Interfaces que elas não utilização, estão adquirindo um alto acoplamento e provavelmente iram sofre problemas quando houver mudanças nos comportamentos das Interfaces.

> O Java coloca a interface como sendo um tipo de referência de dados que contém declarações de métodos, mas sem implementações, sendo, em essência, uma classe abstrata com todos os seus métodos abstratos. Entender o papel que uma interface assume no contexto das aplicações é algo de grande importância. De fato, interfaces proporcionam flexibilidade, permitindo que objetos assumam seus tipos, logo, uma interface pode ser interpretada como sendo simplesmente um papel que um objeto assume em algum momento do seu ciclo de vida. (SILVA, 2013, p. 63)

Segundo Aniche (2015) os problemas de coesão relacionados a Interfaces devem ser tratados da mesma forma como nas classes. Se uma Interface não for coesa devemos dividi-la em duas ou mais, porque ela mais de uma responsabilidade, sendo chamada de “Interface gorda”. Ele destaca que quando este princípio é aplicado corretamente, as Interfaces proporcionam para as classes que as implementam os benefícios de maior reuso e estabilidade.

3.5 - (DIP) - Dependency Inversion Principle (Principle da Inversão de Dependência)

Segundo Silva (2013) este princípio propõe inverter as dependências de uma classe, de forma que ela pare de se acoplar com outras classes concretas e passe a depender de abstrações, este conceito é chamado de acoplamento abstrato. O resultado da aplicação correta do OCP e LSP contribuem na implantação do DIP.

> [Dip2009] justifica a ideia de "inversão" que nomeia o princípio, pelo fato de que modelos mais tradicionais de desenvolvimento, tais como Design e Analise Estruturada, tendem a criar estruturas nas quais módulos de alto nível dependem de módulos de baixo nível e onde abstrações dependem de detalhes, de forma que um dos objetivos desses métodos passa a ser a definição de uma hierarquia de subprograma que descreva como módulos de alto nível fazem chamadas à módulos de nível mais baixo. Nesse sentido, [Dip2009] aponta a estrutura de dependências de um sistema orientado a objetos bem desenhado como sendo "invertida" em relação à estrutura normalmente criada por modelos tradicionais. (SILVA apud DIP, 2013, p. 71/72)

Segundo Aniche (2015), este princípio tem como objetivo principal reduzir ao máximo o acoplamento entre classes, para que seja criado classes mais estáveis que sofram poucas mudanças. Alguns problemas relacionados ao alto acoplamento que o princípio pode corrigir:

- "A partir do momento em que uma classe possui muitas dependências, todas elas podem propagar problemas para a classe principal". (ANICHE, 2015, p. 25)
- "A classe, quando possui muitas dependências, torna-se muito frágil, fácil de quebrar". (ANICHE, 2015, p. 25)

Já se sabe que o acoplamento pode ser um problema para uma classe que se relaciona com muitas outras instáveis. Então, como podemos evita-lo utilizando este princípio? O princípio diz:

- "Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações". (ANICHE, 2015, p. 32)

- "Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações". (ANICHE, 2015, p. 32)

4 – RESULTADOS

Depois de conhecer os conceitos de acoplamento, coesão e princípios SOLID, será apresentado os resultados obtidos a partir da aplicação destes princípios. Como estudo de caso foram criados alguns modelos de diagramas de classes e códigos na linguagem de programação Java, para servir de base de análise na tentativa de identificar se os princípios possuem alguma vantagem ou desvantagem sobre o acoplamento e a coesão.

4 .1 – Análise do Single Responsability Principle

Para melhor entender a importância do SRP, foi utilizado o diagrama da classe (NotaFiscal). Ao observar a classe é possível perceber que ela possui mais de uma responsabilidade: descriminar serviços, salvar dados no Banco de Dados, enviar e-mail e cadastrar clientes, ou seja, uma baixa coesão.

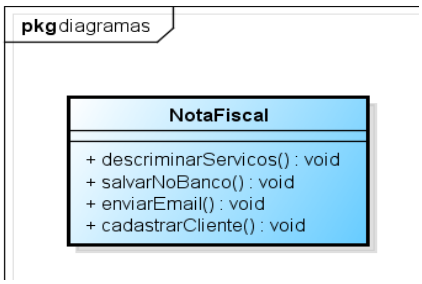

Figura VI - Modelo de diagrama da classe Nota Fiscal
Fonte: Yuri Ramalho Pinheiro, 2016

Para melhorar a implementação e a coesão desta classe, foi aplicado os conceitos do Single Responsability Principle. Conforme propõe o princípio, cada classe deve possuir apenas uma responsabilidade dentro do sistema. Assim a classe (NotaFiscal) teve suas responsabilidades separadas em classe diferentes.

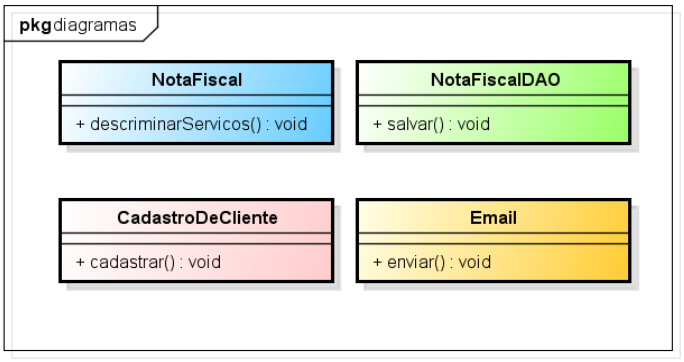

Figura VII - Modelo de diagrama de classe após a aplicação do SRP

Fonte: Yuri Ramalho Pinheiro, 2016

Resultados encontrados:

A partir a aplicação do princípio no diagrama da classe (NotaFiscal), foi possível encontrar os seguintes resultados:

- **Maior coesão**: ficou mais fácil gerenciar o código de cada classe, elas tiveram suas responsabilidades separadas e conforme suas responsabilidades.

- **Menor propagação de erros**: caso ocorra algum erro em alguma das classes, será mais fácil encontra-lo e corrigi-lo, porque cada classe teve sua responsabilidade definida e não está responsável por problemas de outras classes.

- **Sugestão**: é aconselhável utilizar este princípio em conjunto com os outros do SOLID para que haja maior efeito. Também é uma vantagem utiliza-lo na modelagem da camada de negócio dos sistemas, pois ele ajuda a definir como cada regra de negócio deverá ser tratada.

4.2 – Análise do Open Closed Principle

Para melhor entender a importância do OCP, foi utilizado o diagrama de classes abaixo e um trecho de seu código fonte como exemplo.

Observando a classe (CalculoDeSalario) é possível observar que sua responsabilidade é calcular o salário de algum funcionário que será fornecido através da Enumeração (TipoDeFuncionario).

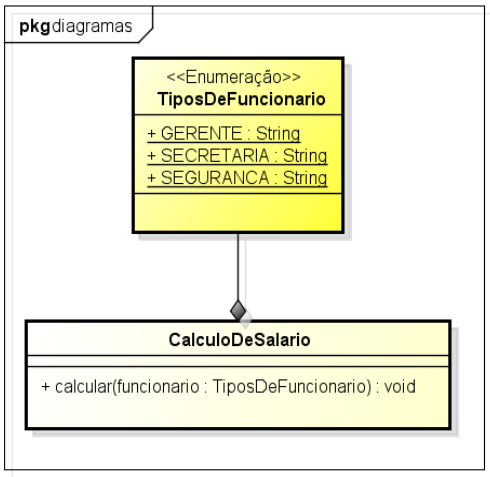

Figura IX - Modelo de Diagrama de classes para efetuar calcula do salário do funcionário

Fonte: Yuri Ramalho Pinheiro, 2016

```
public class CalculoDeSalario {

    public void calcular(TiposDeFuncionario funcionario) {

        if ("FUNCIONARIO".equals(funcionario.GERENTE)) {
            // calcula o salário do GERENTE
        } else if ("FUNCIONARIO".equals(funcionario.SECRETARIA)) {
            // calcula o salário da SECRETARIA
        } else if ("FUNCIONARIO".equals(funcionario.SEGURANCA)) {
            // calcula o salário do SEGURANÇA
        } else {
            // funcinário não existe
            new RuntimeException("Funcionario inválido");
        }
    }
}
```

Figura VIII - Código fonte da classe CalculoDeSalario

Fonte: Yuri Ramalho Pinheiro, 2016

A parti da análise deste modelo de implementação, foi possível observar os seguintes problemas:

- **Baixa coesão:** A classe (CalculoDeSalario) é responsável por calcular o salário de diferentes funcionários e que possuem regras de negócio diferentes.

- **Programação estruturada:** A classe (CalculoDeSario) irá crescer sempre que uma nova regra for adicionada na condição (if/else) para calcular o salário de um novo funcionário, ocasionando em grande aumento do código fonte.

Resultados encontrados:

Após à aplicação do princípio no diagrama acima proposto, foi possível encontrar os seguintes resultados:

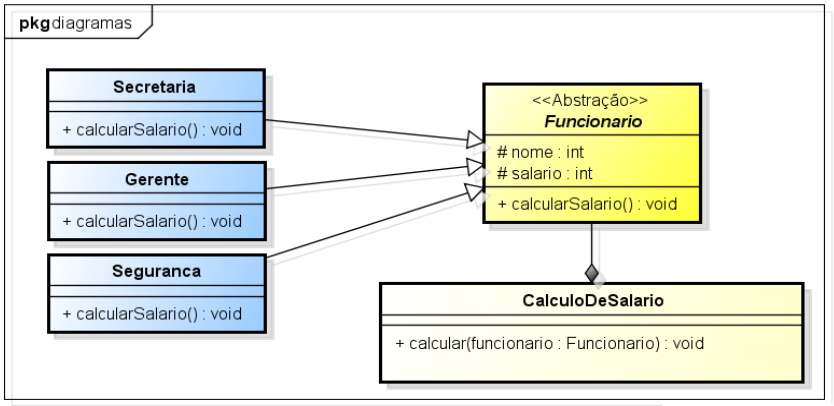

Figura X - Modelo de Diagrama de classes após à aplicação do OCP

Fonte: Yuri Ramalho Pinheiro, 2016

- **Maior coesão**: cada Funcionário foi separado em uma classe diferente que cuidará de suas responsabilidades especificas e a classe (CalculoDeSalario) não irá continuar crescendo.

- **Baixo acoplamento**: a classe Funcionário foi transformada em uma abstração que serve de implementação para suas classes filhas. O relacionamento com a classe (CalculoDeSalario) ficou mais estável.

- **Orientação a objetos**: foi possível utilizar o Polimorfismo no método da classe (CalculoDeSalario) e eliminar a condição (if/else).

- **Sugestão**: Procure utilizar este princípio para utilizar as vantagens da programação orientada a objetos, como o polimorfismo e a herança.

4.3 – Análise do Liskov Substitution Principle

Para demostrar como este princípio pode influenciar a coesão e o acoplamento observe o diagrama abaixo. No relacionamento entre as classes do diagrama é possível perceber que a classe (Cliente) se relaciona com a classe abstrata (Conta) para utilizar os métodos de sacar e depositar. Ela também possui dependência com as classes (ContaPoupanca) e (ContaCorrente), filhas da classe (Conta).

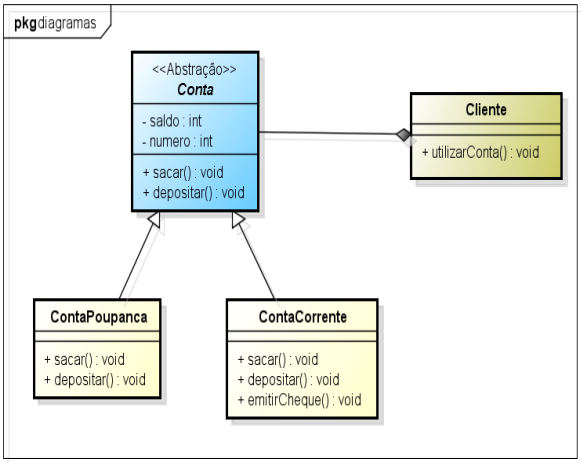

Figura XII - Modelo diagrama de classes gerenciar contas bancárias

Fonte: Yuri Ramalho Pinheiro, 2016

```
3  public class Cliente {
4
5      public void utilizarConta(){
6          Conta conta = new ContaCorrente();
7          conta.depositar("300.00");
8          conta.sacar("50.00");
9          conta.emitirCheque();
10     }
11 }
```

Figura XI - Código fonte da classe Cliente

Fonte: Yuri Ramalho Pinheiro, 2016

Foi observado que quando a classe (Conta) faz referência para sua classe filha (ContaCorrente) e tenta utilizar o método responsável por emitir cheque, os seguintes problemas ocorrem:

- **Propagação de erros:** a classe (Conta) propaga o erro de implementação para a classe (Cliente), quando tenta utilizar o método (emitirCheque) de sua classe filha (ContaCorrente), ocasionando no impedimento da execução do código fonte. Ela infringe o princípio da substituição de Liskov que propõe o seguinte: classes bases (abstratas) devem conhecer as implementações de suas classes filhas.

Como resposta a este problema foi aplicado o LSP. O método (emitirCheque) foi colocado em uma classe separada chamada (Cheque), então a classe (Conta) agora

conhece as implementações de suas classes filhas, porque elas possuem os mesmos métodos (sacar e depositar).

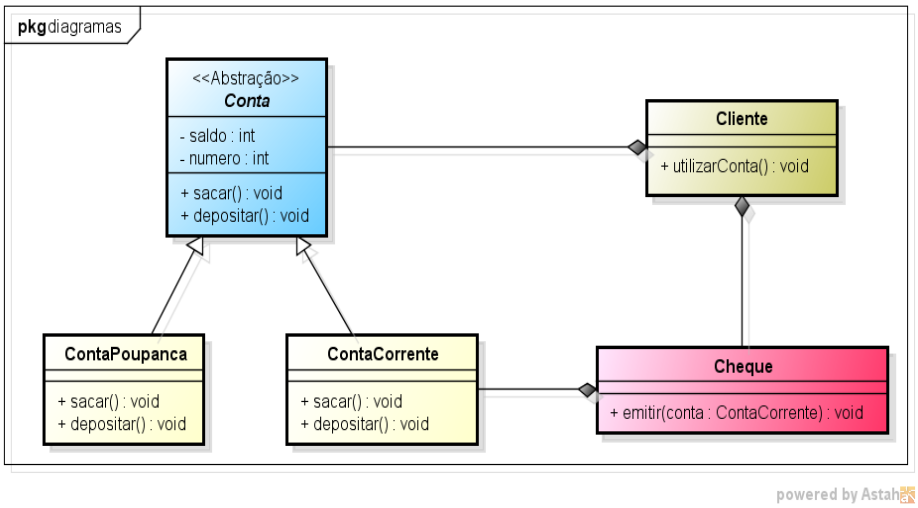

Figura XIII - Modelo de diagrama após a aplicação do LSP

Fonte: Yuri Ramalho Pinheiro, 2016

Resultados encontrados:

A partir a aplicação do princípio no diagrama acima proposto, foi possível encontrar os seguintes resultados:

- **Acoplamento estável**: foi possível evitar a propagação de erros de uma classe para outra e garantir uma melhor estabilidade no relacionamento entre elas, porque o método (emitirCheque) foi separado em outra classe.

- **Alta coesão:** foi possível melhorar as responsabilidades de cada classe.

- **Sugestão:** deve-se ter cuidado ao tentar criar estruturas de classes através de herança, porque uma abstração má planejada pode gerar problemas na implementação das classes e também propagar erros.

4.4 – Análise do Interface Segregation Principle

Neste modelo de diagrama foi proposto duas classes (ISS e IXMX), então uma Interface foi criada e seus comportamentos foram implementados pelas classes.

- ISS: calcula impostos e gera Nota Fiscal.
- IXMX: somente calcula impostos.

O problema deste modelo está na implementação do método (calcularImposto) pela classe (IXMX) que não possui esta responsabilidade. Vale lembrar que quando uma classe implementa uma interface ela é obriga a sobrescrever todos os métodos da interface.

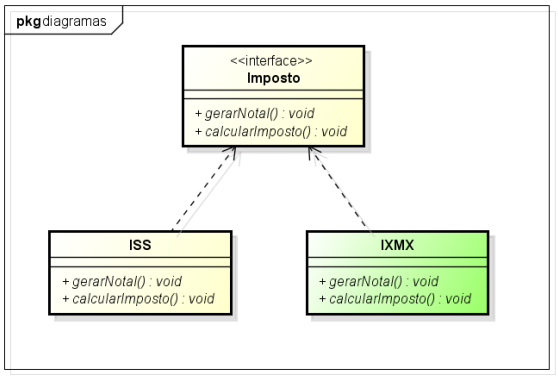

Figura XV - Modelo de diagrama de classes para gerenciar impostos

Fonte: Yuri Ramalho Pinheiro, 2016

```
3  public class IXMX implements Imposto {
4
5      @Override
6      public void calcularImposto(){
7          //lógica para gerar nota
8      }
9
10     @Override
11     public void gerarNota(){
12         // este método não faz nada ou
13         // gera um erro no sistema
14     }
15 }
```

Figura XIV - Código fonte da classe IXMX

Fonte: Yuri Ramalho Pinheiro, 2016

A partir destas observações foi possível encontrar os seguintes problemas:

- **Baixa coesão:** ao implementar o método (gerarNota), a classe (IXMX) recebe a responsabilidade de outra classe, então o método ficará sem sentido nesta classe e ocupando espaço.

- **Propagação de erro:** a Interface (Imposto) força a classe (IXMX) a implementar métodos desnecessários para seu funcionamento.

Após identificado tais problemas, foi aplicado o ISP. O modelo foi modificado e a Interface (Imposto) teve seu método (gerarNota) escrito em outra Interface chamada (NotaFiscal).

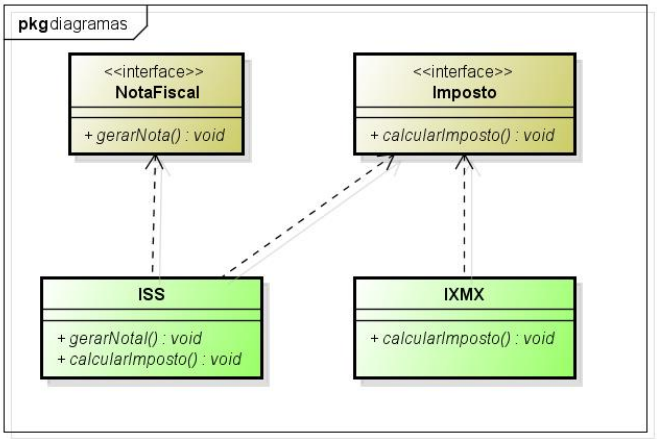

Figura XVI - Modele diagrama de classe após à aplicação do ISP

Fonte: Yuri Ramalho Pinheiro, 2016

Resultados encontrados:

Analisando o novo modelo de diagrama foi possível identificar os seguintes resultados:

- **Alta coesão:** o mesmo conceito que foi utilizado na aplicação do SRP nas classes, foi aplicado para as Interfaces, deixando elas com apenas uma responsabilidade.

- **Baixo acoplamento:** diminui-se o risco de uma classe implementar um método indesejado e deixar o código mais complexo.

- **Sugestão:** este princípio pode ser melhor aplicado se utilizado em conjunto com o SRP. Uma de suas vantagens é evitar que as Interfaces fiquem espalhadas pelo sistema dificultado a manutenção do código. Ele também ajuda criar Interfaces mais especificas.

4.5 – Análise do Dependency Inversion Principle

Para melhor entender a importância deste princípio e sua influência no acoplamento e coesão foi utilizado o seguinte modelo de diagrama de classes. A partir da relação de dependência entre as classes (Cliente) e (Conta) foi observado os seguintes problemas:

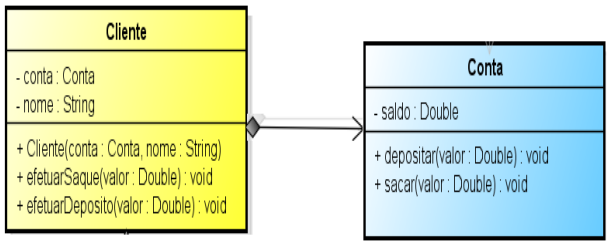

Figura XVII - Modelo de diagrama de classes que gerencia contas bancárias

Fonte: Yuri Ramalho Pinheiro, 2016

- **Acoplamento instável:** a existência da classe (Cliente) dependente diretamente da classe (Conta), porque ela recebe em seu construtor um objeto da classe (Conta).

- **Alto acoplamento:** existe uma relação dependência entre duas classes concretas. Problemas gerados em uma pode ser propagado para a outra.

Após a análise do modelo proposto, foi aplicado o DIP para melhorar a relação de dependência entre as classes. Foi acrescentado uma Interface chamada (Operação) com 2 métodos (sacar e depositar).

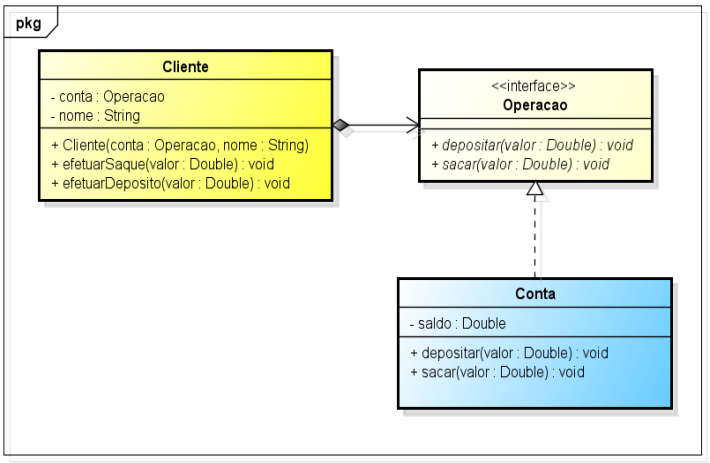

Figura XVIII - Modelo de Diagrama de classes após a aplicação do DIP

Fonte: Yuri Ramalho Pinheiro, 2016

Resultados encontrados:

A partir da implementação deste princípio foi observado os seguintes resultados:

- **Baixo acoplamento:** a classe (Cliente) teve sua dependência invertida, ela deixou de depender da implementação da classe (Conta) e passou a depender da abstração da Interface (Operação). Este novo relacionamento ficou mais flexível e fácil de gerenciar.
- **Acoplamento estável:** caso ocorra algum problema ou alteração na classe (Conta), não será propagado para a classe (Cliente), e vice-versa.

- **Sugestão:** utilize este princípio para tornar o acoplamento entre as classes mais flexível e reutilizável. Ele também é um bom exemplo de como utilizar o polimorfismo da orientação a objetos para implementar mais de um comportamento.

6 - CONCLUSÃO

Após a análise dos resultados obtidos na aplicação do princípio SOLID para gerenciar o acoplamento e coesão das classes de um sistema orientado a objetos, foi possível perceber um positivo ganho de qualidade no planejamento e desenvolvimento dos diagramas de classes e seus códigos. Também se observou que estes princípios são de fundamental importância para aplicar os conceitos da orientação a objetos.

Em consequência do ganho de qualidade após a utilização dos princípios SOLID, é possível concluir que o objetivo proposto foi atingido e estes princípios podem realmente ser uma alternativa de redução do alto acoplamento e baixa coesão. Mas é preciso deixar claro que ela não é única solução, pois durante o percurso desta pesquisa foi observado outras técnicas como os Padrões de Projetos de Software que provavelmente também podem ser uteis, porem este deve ser um estudo para o futuro e possivelmente outra maneira de melhorara o gerenciamento do acoplamento e coesão.

Em suma, manter o alto acoplamento e a baixa coesão é um problema que deve ser evitado quando estamos programando utilizado Orientação a Objetos. Os princípios SOLID foram identificados como uma alternativa de solução para estes problemas e uma maneira correta de utilização da programação Orientada a Objetos.

7 - REFERENCIAS

ANICHE, Mauricio. **Orientação a Objetos e SOLID para Ninjas: Projetando classes flexíveis**. São Paulo, SP: Casa do Código. 2015

ANICHE, Mauricio. **Test Driven Development: Teste e Design no mundo real**. São Paulo, SP: Casa do Código. 2012

CORDEIRO, Gilliard. **CDI: Integre as Dependências e Contexto do seu Código Java**. São Paulo, SP: Casa do Código. 2013

SILVEIRA, Paulo; SILVEIRA, Guilherme; LOPES, Sérgio; MOREIRA, Guilherme; STEPPAT, Nico; KUNG, Fábio. **Arquitetura e Design de Software: Uma visão sobre a plataforma Java**. São Paulo, SP: Casa do Código. 2012

BIBLIOTECA “Newton Paiva”. Desenvolvida por Núcleo de Informática, 2001. **Apresenta produtos e serviços oferecidos pelo Núcleo de Bibliotecas da instituição**. Disponível em: <www.newtonpaiva.br/biblioteca>. Acesso em: 26 set. 2009.

LEANDRO LUQUE. **Coesão e Acoplamento em Sistemas OO**. Disponível emhttps://www.researchgate.net/publication/261026207_Coesao_e_Acoplamento_em_Sistemas_OO -Acesso em: 14 maio. 2016.

EDUARDO PIRES. **Eduardo Pires Treinamento e Consultoria**. Orientação a Objetos - SOLID. Disponível em: http://eduardopires.net.br/2013/04/orientacao-a-objeto-solid/ Acesso em: 12 dez. 2015.

ROBON CASTILHO. **Desenvolvendo Software com Qualidade**. Principio SOLID: Principio de Substituição de Liskov (LSP). Disponível em:

https://robsoncastilho.com.br/2013/03/21/principios-solid-principio-de-substituicao-de-liskov-lsp/ - Acesso em: 12 dez. 2015.

MAURICIO ANICHE. **Princípios do Código SOLID na Orientação o Objetos**.. Disponível em: http://blog.caelum.com.br/principios-do-codigo-solido-na-orientacao-a-objetos/ - Acesso em: 12 dez. 2015.

LEANDRO DANIEL. **Code metrics (parte 3) – Medindo acoplamento**. Disponível em: http://leandrodaniel.com/index.php/code-metrics-parte-3-medindo-acoplamento/ - Acesso em: 12 dez. 2015.

FIROZTAR. **Sun Certified Programmer for Java 6 Study Guide**. Disponível em: http://firozstar.tripod.com/_darksiderg.pdf - Acesso em: 10 maio. 2016.

ALVARO CÉSAR PEREIRA DA SILVA. **Princípios SOLID de Design aplicados na melhoria de códigos fontes em Sistemas Orientados a Objetos**. Disponível em: http://repositorio.ufla.br/bitstream/1/5466/1/MONOGRAFIA_Principios_SOLID_de_design_aplicados_na_melhoria_de_codigo-fonte_em_sistemas_orientados_a_objetos.pdf - Acesso em: 10 maio. 2016.