UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

UNIDADE ACADÊMICA DE EDUCAÇÃO ONLINE

ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

Róger Formagio Lopes

ESTUDO SOBRE PRINCÍPIOS DE DESIGN PARA LINGUAGENS DE PROGRAMAÇÃO ORIENTADAS A OBJETOS VOLTADOS A MANUTENÇÃO DE SISTEMAS DE *SOFTWARE*

São Leopoldo

2019

UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

UNIDADE ACADÊMICA DE EDUCAÇÃO ONLINE

ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

Róger Formagio Lopes

ESTUDO SOBRE PRINCÍPIOS DE DESIGN PARA LINGUAGENS DE PROGRAMAÇÃO ORIENTADAS A OBJETOS VOLTADOS A MANUTENÇÃO DE SISTEMAS DE *SOFTWARE*

Trabalho de Conclusão de Curso apresentado como requisito parcial para a obtenção do título de Especialista em Engenharia de Software, pelo curso de Pós-Graduação Lato Sensu em Engenharia de Software da Universidade do Vale do Rio dos Sinos – UNISINOS.
Orientador: Prof. Ms. Guilherme Silva de Lacerda

São Leopoldo

2019

Estudo sobre princípios de design para linguagens de programação orientadas a objetos voltados a manutenção de sistemas de software

Róger Formagio Lopes[1]

[1]Unidade Acadêmica de Educação Online – Universidade do Vale do Rio dos Sinos (UNISINOS) - São Leopoldo - RS - Brasil

rogerformagiolopes@hotmail.com

Abstract. *This article is a case study of important design principles in the area of object-oriented programming languages aimed at solving maintenance problems in software systems. This study aims to deepen these principles with bibliographic research in order to improve the web system of a real company by passing on the knowledge gained to its team of programmers, but it can also serve any programmer who wants to know these principles better and wants to code better. quality in order to maintain their systems. The main findings of this study indicated that there was a noticeable improvement in the understanding of these principles by the programmer team.*

Resumo. *Este artigo é um estudo de caso sobre importantes princípios de design na área de linguagens de programação orientadas a objetos que visam solucionar problemas de manutenção em sistemas de software. Este estudo visa aprofundar estes princípios com a pesquisa bibliográfica a fim de aprimorar o sistema web de uma empresa real ao repassar o conhecimento adquirido a sua equipe de programadores, mas também pode servir para qualquer programador que queira conhecer melhor estes princípios e almeja codificar com melhor qualidade, visando a manutenção de seus sistemas. Os principais achados deste estudo indicaram que houve uma perceptiva melhora no entendimento destes princípios por parte da equipe de programadores.*

1. Introdução

Normalmente, porém, não necessariamente, a fase de manutenção compreende a maior parte da vida útil de um *software* (SOMMERVILLE, 2011). Apesar disto, os programadores por muitas vezes não se preocupam em criar um código utilizando boas práticas e princípios, com o uso de padrões de projetos conhecidos e com foco na manutenção, e por estas razões acabam criando um código difícil de ser mantido e que logo recebe o título de código legado ou que exige uma reescrita. A manutenção envolve a correção de erros que não foram descobertos em estágios iniciais do desenvolvimento, com melhora da implementação das unidades do sistema e ampliação de seus serviços em resposta às descobertas de novos requisitos (SOMMERVILLE, 2011).

Por muitas vezes, os próprios programadores, codificadores do sistema, não têm controle sobre o impacto das alterações referentes a manutenção dos mesmos, pois acabam por criar um sistema frágil a mudanças. Percebe-se este cenário em muitos projetos, incluindo o projeto da empresa de *software* alvo deste estudo. Por esta razão, a questão desta pesquisa delimita-se em como aprimorar o código desenvolvido pela equipe de desenvolvedores em termos de manutenção e tornar a aplicação destes princípios um hábito dentro da equipe. Estes princípios basicamente de baseiam nos princípios SOLID e GRASP, mas este trabalho também apresenta outros princípios importantes a fim de complementar a pesquisa.

Desenvolver um bom código não é uma tarefa trivial, e pensar em boas práticas, princípios e na manutenibilidade do sistema antes mesmo de desenvolvê-lo é uma tarefa menos trivial ainda. Mas é indispensável um programador conhecer estes princípios, pois somente com conhecimento ele poderá escrever um código de boa qualidade. O código é o legado de uma empresa de *software*, é o produto que a empresa oferece para seus clientes. Não basta escrever um código que funciona momentaneamente, é necessário pensar na sua clareza e em futuras modificações. O software também deve evoluir para atender às necessidades de mudança dos clientes (SOMMERVILLE, 2011).

O objetivo principal desta pesquisa é aprimorar o conhecimento coletivo de uma equipe de programadores de uma empresa real quanto a princípios voltados a manutenção de sistemas, estudados na pesquisa bibliográfica, para que então estes possam escrever códigos melhores em termos de manutenção. Para complementar o estudo e aprimorar este ensino, há também a refatoração de códigos de produção do sistema web desta empresa, que visa a aplicação destes princípios em um cenário real, onde os programadores podem comparar o código atual e o refatorado. Para fixar estes ensinamentos e torna-los uma prática dentro da empresa, este estudo propõe a elaboração de um *checklist* de boas práticas para servir de apoio aos programadores durante a codificação. Espera-se também que este trabalho incentive mais desenvolvedores a estudarem e conhecerem mais sobre princípios de design e padrões de projeto.

Para alcançar este objetivo principal, este trabalho tem como objetivos específicos: *(i)* avaliar princípios de design referentes a linguagens de programação orientadas a objetos voltados a manutenção de sistemas de *software*, como SOLID e GRASP; *(ii)* a refatoração de códigos de produção para a aplicação destes princípios e para complementar o estudo; *(iii)* a elaboração de um *checklist* de boas práticas de codificação para servir de auxílio aos programadores durante o desenvolvimento; *(iv)* a elaboração e realização de um *workshop* com a intuição de passar este conhecimento adquirido aos programadores da equipe alvo; *(v)* a elaboração e realização de um questionário para coleta de *feedback* dos programadores.

Este trabalho está segmentado em 6 seções conforme segue. A seção 1 contém a introdução, a apresentação do tema, a especificação dos problemas de pesquisa, o objetivo principal e os objetivos específicos. A seção 2 contém o referencial teórico e apresenta o resultado da pesquisa bibliográfica sobre princípios voltados a manutenção de sistemas de software. A seção 3 contém os trabalhos relacionados. Na seção 4 encontra-se a metodologia de pesquisa. A seção 5 apresenta o desenvolvimento e análise dos resultados. Por fim, a seção 6 contém as considerações finais.

2. Referencial teórico

O conteúdo desta seção corresponde ao resultado da pesquisa bibliográfica realizada para aprofundar o entendimento dos princípios de design utilizados no desenvolvimento deste estudo. Todos os códigos (Apêndice D) apresentados como exemplo são de fonte do autor desta pesquisa.

2.1. Princípios SOLID

Em meados dos anos 2000, Robert Cecil Martin, popularmente conhecido como *Uncle Bob*, compilou e publicou cinco princípios de desenvolvimento de *software* orientado a objetos, alguns não de sua autoria, em seu artigo (MARTIN, 2000). Estes princípios foram posteriormente chamados de SOLID *principles*, nome sugerido por Michael Feathers, e visam facilitar a expansão, flexibilidade e manutenibilidade de sistemas de *software*. A palavra SOLID é um acrônico para os seguintes princípios:

- *Single-responsibility principle* (SRP), ou princípio da responsabilidade única.
- *Open-closed principle* (OCP), ou princípio de aberto-fechado.
- *Liskov substitution principle* (LSP), ou princípio da substituição de Liskov.
- *Interface segregation principle* (ISP), ou princípio da segregação de interface.
- *Dependency Inversion Principle* (DIP), ou princípio da inversão de dependência.

2.1.1 Princípio da responsabilidade única (SRP)

Uma classe deve possuir somente uma única responsabilidade, podendo haver somente um único motivo pela qual ela mude (MARTIN, 2002, p. 95). Para saber se uma classe possui uma única responsabilidade pode-se pensar em todos os motivos pelo qual ela está sendo alterada. Se há mais de um motivo então se está ferindo o princípio da responsabilidade única. Outra análise que pode ser realizada é verificar se determinado método da classe manipula algum dado dela, ou seja, se alguma informação da classe é necessária para que o método funcione, caso contrário, é possível que este método esteja ferindo o princípio SRP, adicionando à classe uma responsabilidade extra.

O princípio da responsabilidade única está relacionado com o princípio da alta coesão. Uma classe que não é coesa pode apresentar diversos problemas, entre eles:

- Baixa coesão: dificuldade de compreensão e consequentemente dificuldade de manutenção.
- Dificuldade de reuso.
- Fragilidade: com várias responsabilidades entrelaçadas em uma única classe, fica mais difícil alterar uma destas responsabilidades sem comprometer outras.
- Alto acoplamento, ou seja, número excessivo de dependências, tornando-a mais suscetível a alterações em decorrência de alterações em outras classes.

Pode-se pegar como exemplo de violação a seguinte classe:

```
public class Interruptor
{
    public void Ligar()
    {
    }

    public void CalcularPrecoDeVenda()
    {
    }
}
```

Figura 1. Exemplo SRP.

Esta classe possui ao menos duas responsabilidades, ou seja, duas razões para mudar. Uma delas é ligar o interruptor, outra é calcular seu preço de venda. Caso a lógica de ligar o interruptor ou a de calcular o preço de venda mudem, teremos que alterar esta classe, havendo assim mais de um motivo para que ela seja alterada. Este problema pode ser resolvido simplesmente com a criação de outra classe responsável por calcular o preço de venda de produtos.

O princípio da responsabilidade única é um dos mais importantes em linguagens de programação orientada a objetos. Com seu pleno entendimento, consegue-se escrever classes menores, de fácil entendimento, de fácil manutenção e com reuso aprimorado.

2.1.2. Princípio de aberto-fechado (OCP)

Módulos de *software*, ou seja, classes e métodos, devem ser abertos para extensão e fechados para modificação (MARTIN, 2002, p. 99). Em outras palavras, quando é necessário estender o comportamento de um sistema não se deve modificar o código já escrito e sim codificar o novo comportamento apenas criando classes e métodos. Isto pode ser obtido por intermédio da herança, da composição e com o uso de abstrações. Segue um exemplo de violação:

```
public class Arquivo
{
}

public class ArquivoExcel : Arquivo
{
    public void GerarPlanilha()
    {
    }
}

public class ArquivoPdf : Arquivo
{
    public void GerarPdf()
    {
    }
}
```

```
public class GeradorDeArquivos
{
    public void GerarArquivos(List<Arquivo> arquivos)
    {
        foreach (Arquivo arquivo in arquivos)
        {
            switch (arquivo)
            {
                case ArquivoExcel arquivoExcel:
                    arquivoExcel.GerarPlanilha();
                    break;
                case ArquivoPdf arquivoPdf:
                    arquivoPdf.GerarPdf();
                    break;
            }
        }
    }
}
```

Figura 2. Exemplo de violação OCP.

Neste exemplo a classe Arquivo representa um arquivo no sistema operacional e as classes ArquivoExcel e ArquivoPdf estendem o comportamento de Arquivo e fornecem métodos para gerar arquivos em formatos específicos. Há também a classe GeradorDeArquivos, responsável por gerar os arquivos informados por parâmetro.

Agora suponha-se que seja necessário estender a funcionalidade do sistema e adicionar suporte à geração de arquivos tipo texto, no formato txt. Para isto, é necessário que, além de criar uma classe chamada ArquivoTxt, também seja necessário alterar a classe GeradorDeArquivos e adicionar mais uma condição no switch para possibilitar o tratamento de arquivos tipo txt, e este comportamento se repetirá toda vez que seja necessário estender as funcionalidades do *software* para adicionar suporte à novos tipos de arquivo. Pode-se dizer então que a classe GeradorDeArquivos está violando o princípio de aberto-fechado, uma vez que para cada novo tipo de arquivo é necessário modificá-la, ou seja, não está fechada para modificações do tipo “precisa-se de um novo formato de arquivo”.

Neste momento um programador pode pensar qual seria o problema em se adicionar uma simples condição extra ao switch quando surgir um novo formato de arquivo. Os possíveis problemas são:

- Há diversas partes do sistema que fazem uso da geração de arquivos e em cada uma delas há um switch verificando o tipo de arquivo, e para piorar, algumas delas estão em outros componentes. Com isto seria necessário percorrer todo o sistema e componentes para adicionar a nova condição no switch.
- Recompilar e publicar todos os componentes que foram impactados com a alteração.
- Caso se esqueça de alterar uma das partes da aplicação é gerado um erro no sistema.

- Partes do sistema que já foram testadas e já estão em produção estão sendo alteradas. Quanto mais o sistema sofrer mudanças maiores serão as chances de introdução de novos erros.

Mas então como pode-se corrigir estes problemas aplicando o princípio do aberto-fechado? Uma forma melhor de implementação neste caso poderia ser:

```
public interface IGeradorDeArquivo
{
    void GerarArquivo();
}

public class ArquivoExcel : IGeradorDeArquivo
{
    public void GerarArquivo()
    {
    }
}

public class ArquivoPdf : IGeradorDeArquivo
{
    public void GerarArquivo()
    {
    }
}

public class GeradorDeArquivos
{
    public void GerarArquivos(List<IGeradorDeArquivo> geradoresDeArquivo)
    {
        foreach (IGeradorDeArquivo geradorDeArquivo in geradoresDeArquivo)
        {
            geradorDeArquivo.GerarArquivo();
        }
    }
}
```

Figura 3. Exemplo OCP com interfaces.

Os passos realizados foram os seguintes:

- Criação da interface IGeracaoDeArquivo e do método GerarArquivo.
- Implementação da interface IGeracaoDeArquivo nas classes ArquivoExcel e ArquivoPdf.
- Na classe GeradorDeArquivos, foi alterado o tipo de parâmetro do método GerarArquivos para List<IGeradorDeArquivo>, eliminando a checagem de tipo de Arquivo e passando-se a utilizar polimorfismo.

Com a utilização desta técnica, não importa quantos tipos extras de arquivo o sistema necessite implementar, não será necessário alterar a classe GeradorDeArquivos, respeitando assim o princípio aberto-fechado. Para cada novo tipo de arquivo que necessite gerar um arquivo, basta implementar a interface IGeradorDeArquivo, ou seja, o *software* está assim aberto para extensão e fechado para modificação.

Além do uso de interfaces, pode-se também fazer uso da herança:

```
public abstract class Arquivo
{
    public abstract void Gerar();
}

public class ArquivoExcel : Arquivo
{
    public override void Gerar()
    {
    }
}

public class ArquivoPdf : Arquivo
{
    public override void Gerar()
    {
    }
}

public class GeradorDeArquivos
{
    public void GerarArquivos(List<Arquivo> arquivos)
    {
        foreach (Arquivo arquivo in arquivos)
        {
            arquivo.Gerar();
        }
    }
}
```

Figura 4. Exemplo OCP com herança.

Os passos realizados aqui foram os seguintes:

- Arquivo tornou-se uma classe abstrata.
- Foi criado um método abstrato chamado Gerar.
- Cada classe herdada agora implementa o método Gerar.
- Na classe GeradorDeArquivos, eliminou-se a checagem de tipo de Arquivo e passou-se a utilizar polimorfismo.

Embora se esteja observando e aplicando o princípio aberto-fechado no exemplo acima, é necessário observar que o uso de herança pode acarretar em problemas de *design* e por esta razão seu uso precisa ser bem analisado antes de sua escolha. Por exemplo: caso seja necessário implementar a remoção de arquivos, pode-se simplesmente criar um novo método chamado Apagar na classe abstrata, forçando as classes derivadas a implementá-lo. Porém, e se existir, por alguma razão, algum tipo de arquivo que não pode ser apagado? Como esta exceção deve ser tratada? Seria disparada uma exceção no sistema? Seria tratado o tipo de arquivo a fim de verificar se o método Apagar pode ser invocado, violando assim novamente o princípio de aberto-fechado? O método simplesmente não faria nada? Todas estas soluções não são consideradas boas práticas. É preferível o *software* ter o comportamento correto do que criar formas de contornar problemas em seu *design*.

2.1.3 Princípio da substituição de Liskov (LSP)

Este princípio possui este nome por ter sido criado por Bárbara Liskov. Ele foi posteriormente resumido e popularizado por Robert Cecil Martin em seu livro (MARTIN, 2002). Basicamente, diz que qualquer classe derivada deve ser substituível por sua classe base sem mudar o seu comportamento ou o funcionamento correto do programa (MARTIN, 2002, p. 111).

A seguir é visto um exemplo bem simples de violação de LSP para um melhor entendimento deste conceito, baseado no exemplo do princípio aberto-fechado:

```
public class Arquivo
{
}

public class ArquivoExcel : Arquivo
{
    public void GerarPlanilha()
    {
    }
}

public class ArquivoPdf : Arquivo
{
    public void GerarPdf()
    {
    }
}
```

Figura 5. Exemplo de violação LSP.

Pode-se observar que as classes ArquivoExcel e ArquivoPdf herdam da classe Arquivo, possivelmente para o reaproveitamento de campos e comportamentos. Porém, cada uma destas classes possui métodos distintos, impedindo o uso de polimorfismo. E é exatamente por isto que a classe GeradorDeArquivos necessita fazer um *downcast* para poder invocar os métodos de geração de arquivo:

```
public class GeradorDeArquivos
{
    public void GerarArquivos(List<Arquivo> arquivos)
    {
        foreach (Arquivo arquivo in arquivos)
        {
            switch (arquivo)
            {
                case ArquivoExcel arquivoExcel:
                    arquivoExcel.GerarPlanilha();
                    break;
                case ArquivoPdf arquivoPdf:
                    arquivoPdf.GerarPdf();
                    break;
            }
        }
    }
}
```

Figura 6. Exemplo de violação LSP.

Desta forma se está ferindo o princípio LSP, uma vez que não é possível substituir as classes derivadas por sua classe base Arquivo, para poder invocar os métodos de geração de arquivo. Outra observação importante é que ao ferir LSP também se fere OCP.

O exemplo acima é um exemplo bem simples, onde as classes derivadas não permitem o polimorfismo por não possuírem os mesmos métodos da classe base. Mas existem também exemplos bem mais sutis, de quebra de comportamento, como o conhecido "Quadrado é um retângulo":

```
public class Retangulo
{
    public virtual int Altura { get; set; }
    public virtual int Largura { get; set; }

    public int CalcularArea()
    {
        return Altura * Largura;
    }
}

public class Quadrado : Retangulo
{
    public override int Altura
    {
        get { return base.Altura; }
        set { base.Altura = base.Largura = value; }
    }

    public override int Largura
    {
        get { return base.Altura; }
        set { base.Altura = base.Largura = value; }
    }
}
```

Figura 7. Exemplo de violação LSP.

É definida uma classe Retangulo com as propriedades Altura e Largura e um método para realizar o cálculo da área. Também é definida uma classe Quadrado derivada de Retangulo, afinal um quadro é um tipo de retângulo. O problema apresentado aqui é uma quebra de comportamento, uma vez que se sabe que um retângulo pode ter sua altura e largura alterados invariavelmente, desde que não sejam iguais, que ele permanecerá um retângulo. Porém, um quadrado precisa sobrescrever este comportamento e manter sua altura e largura sempre iguais para permanecer um quadrado, ou seja, a classe derivada está violando o comportamento da classe base. Este problema se torna aparente em situações como:

```
public void MetodoQualquer(Retangulo retangulo)
{
    retangulo.Altura = retangulo.Altura * 2;
    retangulo.Largura = retangulo.Largura * 4;

    int area = retangulo.CalcularArea();
    // Realiza alguma operação com a área...
}
```

Figura 8. Exemplo de violação LSP.

Este método recebe um retângulo qualquer, duplica sua altura, quadriplica sua largura e realiza alguma operação com a área calculada. O programador assumiu que estava lidando com um retângulo e aplicou um cálculo que alterasse suas dimensões. O problema é que caso este método receba um Quadrado, o que é possível pois um Quadrado é um Retangulo, gera-se um comportamento inesperado: ao alterar sua largura também se está alterando sua altura, e vice-versa, tornando o retângulo em um quadrado. Ou seja, o comportamento da classe base está sendo alterado, ferindo o princípio de substituição de Liskov, e a alteração de comportamento de objetos derivados pode gerar situações inesperadas e erros durante a execução do programa. O problema se agrava à medida que mais métodos e propriedades forem definidos na classe base.

2.1.4. Princípio da segregação de interface (ISP)

Clientes não devem ser forçados a implementar interfaces que não usam ou não devem ser forçados a depender de métodos que não usam (MARTIN, 2002, p. 135). Em outras palavras, é melhor utilizar interfaces específicas e mais especializadas do que uma interface genérica. Muitas vezes, se começa a pensar em interfaces ao ver um padrão ou em tarefas repetitivas. Porém, em muitas destas vezes se acaba por criar uma interface genérica, forçando o cliente a implementar métodos desnecessariamente.

Como exemplo, suponha-se que para desenvolver um programa simples relacionado a pássaros deve-se criar uma interface simples chamada IPassaro contendo os métodos Cantar e Voar, afinal, estas características são comuns em pássaros:

```
public interface IPassaro
{
    void Cantar();
    void Voar();
}
```

Figura 9. Exemplo de violação ISP.

Porém, durante uma reunião com o cliente este decide que o programa deve também ter uma avestruz, afinal uma avestruz também é um tipo de pássaro:

```
public class Avestruz : IPassaro
{
    public void Cantar()
    {
        // Reproduz o canto da avestruz
    }

    public void Voar()
    {
        // Não faz nada pois avestruz não pode voar
    }
}
```

Figura 10. Exemplo de violação ISP.

Como a classe Avestruz implementa a interface IPassaro, ela é forçada a implementa o método Voar. Porém, como uma avestruz não pode voar, este método acaba não fazendo nada. Uma forma simples de resolver esta questão é segregar a interface IPassaro e criar interfaces menos genéricas que simbolizam os comportamentos destes animais:

```
public interface IPassaro
{
}

public interface IPassaroCantor
{
    void Cantar();
}

public interface IPassaroVoador
{
    void Voar();
}
```

Figura 11. Exemplo ISP.

Segregou-se os comportamentos de pássaros em duas interfaces: ICantor e IVoador. Desta forma, pode-se atribuir comportamentos aos pássaros de formas distintas para cada pássaro:

```
public class BeijaFlor : IPassaro, IPassaroCantor, IPassaroVoador
{
    public void Cantar()
    {
        // Reproduz o canto do beija-flor
    }

    public void Voar()
    {
        // Realiza vôo como um beija-flor
    }
}

public class Avestruz : IPassaro, IPassaroCantor
{
    public void Cantar()
    {
        // Reproduz o canto da avestruz
    }
}
```

Figura 12. Exemplo ISP.

Deve-se observar, porém, que nem sempre a implementação de várias interfaces em uma classe é o melhor caminho, pois isto impede o reuso de código em algumas situações. Por exemplo, caso a forma de reproduzir o canto dos pássaros mude, é necessário também mudar todas as implementações deste comportamento em cada classe. Por esta razão, recomenda-se uma análise prévia destes fatores quanto aos requisitos do sistema. O item “Favorecer composição sobre herança” deste estudo traz uma abordagem diferente para este problema.

2.1.5. Princípio da inversão de dependência (DIP)

O princípio DIP fala sobre duas regras: a primeira delas é que se deve programar para uma interface e não para uma implementação (MARTIN, 2002, p. 127), ou seja, módulos devem depender de abstrações e não de classes concretas. A segunda delas diz que módulos de alto nível não devem depender de módulos de baixo nível e que ambos devem depender de abstrações (MARTIN, 2002, p. 127). O seguinte exemplo viola DIP:

```
public class GerenciadorDeTransferencia
{
    public void Transferir(Pendrive pendrive, DiscoRigido discoRigido)
    {
        Byte[] dados = pendrive.LerDados();
        discoRigido.GravarDados(dados);
    }
}

public class Pendrive
{
    public byte[] LerDados()
    {
        // Realiza a leitura dos dados do pendrive
    }
}

public class DiscoRigido
{
    public void GravarDados(byte[] dados)
    {
        // Realiza a gravação dos dados no disco rígido
    }
}
```

Figura 13. Exemplo de violação DIP.

A classe GerenciadorDeTransferencia é considerada um módulo de alto nível pois define a lógica de como os dados são transferidos de um dispositivo para outro, e para isto ela utiliza as classes Pendrive e DiscoRigido, consideradas módulos de baixo nível. Além disto, ela também possui uma forte dependência das classes concretas Pendrive e DiscoRigido, o que resulta em uma arquitetura frágil a mudanças, e uma arquitetura frágil a mudanças de seus módulos se torna cada vez mais difícil de ser mantida com a evolução do sistema. E caso se tenha mais um dispositivo para realizar uma transferência, como um HD externo por exemplo, é necessário modificar o módulo de alto nível.

Pensando-se no princípio da inversão de dependência, pode-se aplicar as seguintes mudanças:

```
public interface IGerenciadorDeTransferencia
{
    void Transferir(IUnidadeExterna unidadeExterna, IUnidadeInterna unidadeInterna);
}

public interface IUnidadeInterna
{
    void GravarDados(byte[] dados);
}

public interface IUnidadeExterna
{
    byte[] LerDados();
}
```

```
public class GerenciadorDeTransferencia : IGerenciadorDeTransferencia
{
    public void Transferir(IUnidadeExterna unidadeExterna, IUnidadeInterna unidadeInterna)
    {
        byte[] dados = unidadeExterna.LerDados();
        unidadeInterna.GravarDados(dados);
    }
}

public class Pendrive : IUnidadeExterna
{
    public byte[] LerDados()
    {
        // Realiza a leitura dos dados do pendrive
    }
}

public class DiscoRigido : IUnidadeInterna
{
    public void GravarDados(byte[] dados)
    {
        // Realiza a gravação dos dados no disco rígido
    }
}
```

Figura 14. Exemplo DIP.

Agora o módulo de alto nível GerenciadorDeTransferencia não depende mais de módulos de baixo nível. Além disto, ele não depende mais de classes concretas e sim de abstrações. Desta forma, pode-se criar inúmeros módulos de baixo nível sem precisar alterar o módulo de alto nível.

2.2 Princípios GRASP

GRASP (LARMAN, 2005) é o acrônimo para *General Responsibility Assignment Software Patterns* (ou *Principles*), ou seja, é um conjunto de princípios para atribuição de responsabilidades para classes e objetos dentro de um projeto que utiliza uma linguagem orientada a objetos. Da mesma forma que o SOLID, são diretrizes que visam aplicar padrões bem consolidados de programação na construção de objetos, tornando-os mais organizados, de fácil manutenção e expansão.

Ao todo são nove os padrões e princípios pertencentes ao GRASP: alta coesão (*high cohesion*), baixo acoplamento (*low coupling*), polimorfismo (*polimorphism*), criador (*creator*), controlador (*controller*), especialista na informação (*information expert*), fabricação pura (*pure fabrication*), indireção (*indirection*) e variações protegidas (*protected variations*).

2.2.1. Alta coesão

A coesão está ligada ao princípio da responsabilidade única, um dos princípios do SOLID introduzidos por Robert Cecil Martin detalhado anteriormente neste estudo, e diz que uma classe deve ter uma única responsabilidade, ou seja, um único motivo para ser alterada. Se diz que uma classe possui uma alta coesão quando esta possui uma única responsabilidade, e baixa coesão quando possui múltiplas responsabilidades. Em uma classe com baixa coesão podem ocorrer problemas como:

- Difícil de entender.
- Difícil de reusar.
- Difícil de manter.
- Torna-se frágil, necessitando alterações quando as demais classes forem alteradas.

2.2.2. Baixo acoplamento

O acoplamento define o grau de dependência de uma classe quanto à outras classes, ou seja, é uma medida de quão fortemente uma classe está conectada, possui conhecimento ou depende de outras classes. Se uma classe depende de poucas classes, ela possui um baixo acoplamento, e se depende de muitas classes ela possui um alto acoplamento. Um alto acoplamento pode trazer problemas como:

- Mudanças nas classes relacionadas podem forçar mudanças locais à classe.
- A classe apresenta dificuldades de compreensão.
- Torna-se mais difícil o reuso da classe.

De certa forma, o baixo acoplamento e o princípio da responsabilidade única são opostos, pois, ao dividir uma classe em mais classes, cada uma com sua responsabilidade, se está aumentando o acoplamento. Porém, a ideia do *low coupling* é manter o menor acoplamento possível e não o eliminar, afinal, na programação orientada a objetos dificilmente não haverá relacionamentos entre classes.

2.2.3. Polimorfismo

Polimorfismo é um recurso básico em uma linguagem de programação orientada à objetos. Ele permite que classes derivadas possam ser referenciadas por suas classes base e que métodos de uma classe base possam invocar métodos que possuem a mesma assinatura, porém com comportamentos distintos, em suas classes derivadas. A decisão sobre qual método deve ser executado é realizada em tempo de execução. Pode-se citar como exemplo o conceito aberto-fechado utilizado anteriormente:

```
public abstract class Arquivo
{
    public abstract void Gerar();
}

public class ArquivoExcel : Arquivo
{
    public override void Gerar()
    {
        // Gera arquivo em excel
    }
}

public class ArquivoPdf : Arquivo
{
    public override void Gerar()
    {
        // Gera arquivo em pdf
    }
}
```

```
public class GeradorDeArquivos
{
    public void GerarArquivos(List<Arquivo> arquivos)
    {
        foreach (Arquivo arquivo in arquivos)
        {
            arquivo.Gerar();
        }
    }
}
```

Figura 15. Exemplo polimorfismo.

Graças ao polimorfismo, o método GerarArquivos da classe GeradorDeArquivos aceita uma lista de qualquer classe derivada de Arquivo, neste caso ArquivoExcel e ArquivoPdf, e ao ser invocado o método Gerar, é decidido em tempo de execução qual método será disparado, podendo ser tanto o método Gerar da classe ArquivoExcel como o da classe ArquivoPdf. Isto dependerá da instância do objeto.

O GRASP provê boas práticas com o uso do polimorfismo dentro de um projeto que utiliza uma linguagem de programação orientada a objetos, e estas boas práticas estão relacionadas com os princípios aberto-fechado e inversão de dependência do SOLID, pois ao se aplicar polimorfismo, também está se aplicando as boas práticas exploradas por estes princípios:

- Depender de abstrações e não de classes concretas.
- Tornar o código menos suscetível a mudanças de outros módulos.

2.2.4. Criador

O padrão *creator* é um princípio que ajuda na decisão sobre qual classe deve ser responsável pela criação de novas instâncias de objetos. Sabe-se que a criação de instâncias de objetos é um dos recursos mais utilizados na programação orientada à objetos e por esta razão é útil ter um princípio dedicado a esta finalidade. Seguindo este princípio, as classes do sistema apresentarão maior clareza, menor acoplamento, melhor encapsulamento e melhor reutilização.

Basicamente, este princípio indica que deve ser atribuída à classe B a responsabilidade de criar uma instância da classe A caso uma das seguintes premissas forem verdadeiras:

- B agrega objetos da classe A.
- B contém objetos da classe A.
- B registra instâncias da classe A.
- B usa muito objetos da classe A.
- B possui os dados necessários para inicializar A.

Um dos usos mais comuns do *creator* é juntamente a uma *Factory*, para encapsular a criação de instâncias de objetos. Assim, o objeto consumidor apenas decide qual instancia ele quer utilizar. Outro uso comum é junto ao padrão de inversão de dependência, criando assim uma camada de abstração intermediaria que fica responsável por criar as instâncias dos objetos.

Para exemplificar o resultado da não utilização do *creator*, pode-se pensar no seguinte código de exemplo, onde há a classe Pedido que possui uma lista de PedidoItem, criado a partir de Produto:

```
public class Pedido
{
    public List<PedidoItem> Itens { get; private set; }

    public Pedido()
    {
        Itens = new List<PedidoItem>();
    }
}

public class PedidoItem
{
    private readonly string nome;
    private readonly double precoUnitario;
    private readonly int quantidade;

    public PedidoItem(Produto produto)
    {
        nome = produto.Nome;
        precoUnitario = produto.Preco;
        quantidade = 1;
    }
}

public class Produto
{
    public string Nome { get; set; }
    public double Preco { get; set; }
}
```

Figura 16. Exemplo de violação criador.

Agora um exemplo de utilização desta estrutura de classes:

```
// Criação da instância de pedido
var pedido = new Pedido();

// Criação de um produto
var produto = new Produto()
{
    Nome = "Pastel assado de frango",
    Preco = 2.50
};

// Criação do pedido item, violando o encapsulamento
var pedidoItem = new PedidoItem(produto);

// É exposta a implementação interna do objeto pedido
pedido.Itens.Add(pedidoItem);
```

Figura 17. Exemplo de violação criador.

Pode-se perceber que com esta estrutura é necessário criar um item de pedido manualmente e então inseri-lo diretamente na lista interna de pedidos, violando assim o encapsulamento e expondo a implementação interna do objeto pedido. Repensando esta estrutura de classes com a utilização do *creator*, pode-se realizar as seguintes mudanças:

```
public class Pedido
{
    private readonly List<PedidoItem> itens;

    public Pedido()
    {
        itens = new List<PedidoItem>();
    }

    public void CriarNovoPedidoItem(Produto produto)
    {
        itens.Add(new PedidoItem(produto));
    }
}

public class PedidoItem
{
    private readonly string nome;
    private readonly double precoUnitario;
    private readonly int quantidade;

    public PedidoItem(Produto produto)
    {
        nome = produto.Nome;
        precoUnitario = produto.Preco;
        quantidade = 1;
    }
}

public class Produto
{
    public string Nome { get; set; }
    public double Preco { get; set; }
}
```

Figura 18. Exemplo criador.

Pode-se observar que agora a classe responsável por criar uma instância de PedidoItem é a classe Pedido, pois possui os dados necessários para criar instâncias de PedidoItem, que no caso é o Produto. Agora com o padrão *creator* implementado a utilização da arquitetura fica da seguinte forma:

```
// Criação da instância de pedido
var pedido = new Pedido();

// Criação de um produto
var produto = new Produto()
{
    Nome = "Pastel assado de frango",
    Preco = 2.50
};

// É solicitamos ao objeto pedido a criação de um novo pedido item
pedido.CriarNovoPedidoItem(produto);
```

Figura 19. Exemplo criador.

Também pode-se observar que agora não é mais necessário se preocupar em como um novo item de pedido é criado, basta adicionar os itens ao pedido.

2.2.5. Especialista da informação

Este princípio oferece instruções sobre qual classe se deve atribuir uma responsabilidade. Basicamente, ele diz que uma responsabilidade A deve ser atribuída à classe B se B possui as informações necessárias para que a responsabilidade A seja executada, ou seja, B seria o especialista da informação. Pode-se ter como exemplo o exercício anterior exemplificado no padrão *creator*:

```
public class Pedido
{
    private readonly List<PedidoItem> itens;

    public Pedido()
    {
        itens = new List<PedidoItem>();
    }

    public void CriarNovoPedidoItem(Produto produto)
    {
        itens.Add(new PedidoItem(produto));
    }
}

public class PedidoItem
{
    private readonly string nome;
    private readonly double precoUnitario;
    private readonly int quantidade;

    public PedidoItem(Produto produto)
    {
        nome = produto.Nome;
        precoUnitario = produto.Preco;
        quantidade = 1;
    }
}

public class Produto
{
    public string Nome { get; set; }
    public double Preco { get; set; }
}
```

Figura 20. Exemplo especialista da informação.

Supõe-se que seja necessário calcular o valor total do pedido. A quem se deve atribuir a responsabilidade de realizar este cálculo? Segundo o princípio *information expert*, deve-se averiguar quem possui as informações necessárias para tal. Neste caso, é a classe Pedido que possui os itens do pedido necessários para o cálculo:

```
public class Pedido
{
    private readonly List<PedidoItem> itens;

    public Pedido()
    {
        itens = new List<PedidoItem>();
    }

    public void CriarNovoPedidoItem(Produto produto)
    {
        itens.Add(new PedidoItem(produto));
    }

    public double CalcularValorTotal()
    {
        return itens.Sum(item => item.Quantidade * item.PrecoUnitario);
    }
}
```

Figura 21. Exemplo especialista da informação.

Criou-se então um método CalcularValorTotal dentro da classe Pedido que retorna a soma da quantidade vezes o preço unitário de cada item.

2.2.6. Controlador

O controlador é um objeto responsável por tratar eventos ou definir métodos para as operações do sistema. Um evento do sistema é um evento de alto nível gerado por um ator externo, como um usuário clicando em um botão da interface por exemplo. O padrão *controller* serve para minimizar a dependência entre a interface de usuário e as classes de modelo de domínio. O *controller* é o primeiro objeto após a camada de interface de usuário que trata requisições de operação do sistema e delega a responsabilidade para objetos de domínio de níveis mais baixos. Um controlador pode representar:

- Todo o sistema, um dispositivo ou um subsistema, sendo assim chamado de controlador de fachada ou *facade controller*.
- Um cenário de caso de uso, como por exemplo ControladorDe<NomeDoCasoDeUso>.

Um controlador de fachada é o ponto principal para todas as chamadas provenientes da interface do usuário ou outros sistemas, sendo adequado quando não há uma quantidade muito grande de eventos do sistema ou quando não é possível para a interface de usuário redirecionar mensagens de eventos de sistema para controladores alternativos. Um exemplo de um controlador de fachada seria:

```
public class ControladorTerminalPortatilVendas
{
    public void IniciarNovaVenda()
    {
        // Implementação
    }

    public void InserirItem()
    {
        // Implementação
    }

    public void FinalizarVenda()
    {
        // Implementação
    }

    public void RealizarPagamento()
    {
        // Implementação
    }
}
```

Figura 22. Exemplo controlador de fachada.

Pode-se observar que este controlador de fachada trata as ações desde o início até o fim de um terminal portátil de vendas (TPV), mas ainda assim é um controlador com poucas ações.

Um controlador de caso de uso é diferente para cada caso de uso e é uma alternativa quando os controladores de fachada deixarem a classe com muitas responsabilidades (alto acoplamento e/ou baixa coesão). Um exemplo de controlador de caso de uso poderia ser a manutenção de clientes no sistema:

```
public class ControladorCliente
{
    public void Listar()
    {
        // Implementação
    }

    public void Incluir()
    {
        // Implementação
    }

    public void Editar()
    {
        // Implementação
    }

    public void Excluir()
    {
        // Implementação
    }
}
```

Figura 23. Exemplo controlador de caso de uso.

2.2.7. Fabricação pura

Sabe-se que os princípios de alta coesão e baixo acoplamento são um dos pilares do desenvolvimento em uma linguagem orientada à objetos. Porém, algumas situações forçam a diminuição da coesão, princípio ligado ao princípio da responsabilidade única (SRP), quando se tenta seguir alguns princípios como o de especialista da informação (*information expert*).

Supõe-se por exemplo que exista uma classe de domínio relacionada à clientes chamada ServicoCliente e que agora é necessário adicionar suporte à persistência destes clientes em um banco de dados relacional. O princípio especialista da informação diz que esta persistência deve ser implementada na própria classe ServicoCliente, pois é ela que contém as informações sobre os clientes. Porém, persistir dados em um banco de dados requer várias etapas como por exemplo abrir uma conexão, acessar a tabela, salvar os dados, etc, responsabilidades estas que não condizem com a classe de domínio ServicoCliente, diminuindo assim a coesão e aumentando o acoplamento desta classe que agora passaria a ter mais de uma responsabilidade e precisaria conhecer vários outros objetos relacionados à banco de dados. Além disto, para cada classe que precisar persistir seus dados seria necessário replicar boa parte do código, ou seja, sem reaproveitamento.

É exatamente aí que entra o princípio da fabricação pura, também chamado de princípio de invenção pura. Ele diz que em uma situação como esta, onde acaba-se diminuindo a coesão e aumentando o acoplamento, é necessário criar uma classe que seja unicamente responsável pela persistência dos dados em um objeto de persistência, como um banco de dados relacional por exemplo. Esta nova classe então seria uma fabricação pura, ou invenção pura, inventada com a finalidade de diminuir o acoplamento da classe ServicoCliente e aumentar sua coesão.

2.2.8. Indireção

Este princípio é similar ao princípio da fabricação pura, uma vez que tenta solucionar o problema de alto acoplamento entre classes. Como permitir que dois objetos interajam entre si, mas evitando que haja um acoplamento direto entre eles? Como desacoplar objetos apoiando o baixo acoplamento e maximizando o potencial de reuso? A resposta é a criação de um objeto intermediário que recebe esta responsabilidade. O objeto intermediário cria uma camada de indireção entre objetos que não dependem mais um do outro e sim somente da indireção.

Um exemplo de indireção é a utilização do controlador (*controller*) no padrão MVC (*Model-View-Controller*). O controlador cria uma camada de indireção ao ser o intermediador entre o modelo de dados (*model*) e a camada de apresentação (*view*).

2.2.9. Variações protegidas

Como atribuir responsabilidades a objetos, subsistemas e sistemas de modo que as variações ou instabilidades nestes elementos não tenham um impacto indesejável sobre outros elementos? De fato, variações em sistemas são umas das principais razões para o surgimento de erros e comportamentos inesperados, pois em muitos casos é difícil averiguar a extensão dos impactos que uma alteração pode causar.

Para proteger variações é necessário primeiramente identificar os pontos de *software* que variam ou que possam sofrer uma instabilidade prevista e então separar estas variantes das partes que permanecem inalteradas em abstrações, de modo que agora o sistema dependa de uma abstração e não mais de algo concreto, criando assim um nível de indireção. Por intermédio desta abstração pode-se assim criar várias implementações para cada tipo de variação, e graças ao polimorfismo será decidido em tempo de execução qual implementação será utilizada. O encapsulamento de objetos, ou seja, objetos internos de classes, também colaboram para proteção de variações. Eis um exemplo prático:

```
public enum TipoMovimentacaoFinanceira
{
    CONTA_PAGAR = 1,
    CONTA_RECEBER = 2,
    COBRANCA = 3
}

public class ContaPagar
{
    public void GerarContaPagar(double valor)
    {
    }
}

public class ContaReceber
{
    public void GerarContaReceber(double valor)
    {
    }
}

public class Cobranca
{
    public void GerarCobranca(double valor)
    {
    }
}
```

```
public class Financeiro
{
    private void GerarMovimentacaoFinanceira(double valor, TipoMovimentacaoFinanceira
tipo)
    {
        switch (tipo)
        {
            case TipoMovimentacaoFinanceira.CONTA_PAGAR:
                new ContaPagar().GerarContaPagar(valor);
                break;
            case TipoMovimentacaoFinanceira.CONTA_RECEBER:
                new ContaReceber().GerarContaReceber(valor);
                break;
            case TipoMovimentacaoFinanceira.COBRANCA:
                new Cobranca().GerarCobranca(valor);
                break;
        }
    }
}
```

Figura 24. Exemplo de violação variações protegidas.

Pode-se observar que há uma cláusula *switch* que escolhe em tempo de execução qual classe será chamada para tratar a movimentação financeira. O problema com esta abordagem é que, além de estar gerando um alto acoplamento entre a classe Financeiro e várias outras classes, também se está criando um trecho de código que sofrerá variações no futuro, a medida que mais tipos de movimentação financeira forem suportados pelo sistema. Além disto, é possível que esta cláusula *switch* se repita em vários trechos de código espalhados pelo sistema e subsistemas, de modo que, ao adicionar um novo tipo de movimentação seja necessário também alterar todas as ocorrências deste *switch*, aumentado assim as chances de ocorrerem bugs ou comportamentos inesperados, além de alterar trechos de código já testados ou que estejam em produção. Para proteger estas variações pode-se redefinir o exemplo da seguinte forma:

```
public enum TipoMovimentacaoFinanceira
{
    CONTA_PAGAR = 1,
    CONTA_RECEBER = 2,
    COBRANCA = 3
}

public interface IMovimentacaoFinanceira
{
    void GerarMovimentacaoFinanceira(double valor);
}

public class ContaPagar : IMovimentacaoFinanceira
{
    public void GerarMovimentacaoFinanceira(double valor)
    {
    }
}
```

```
public class ContaReceber : IMovimentacaoFinanceira
{
    public void GerarMovimentacaoFinanceira(double valor)
    {
    }
}

public class Cobranca : IMovimentacaoFinanceira
{
    public void GerarMovimentacaoFinanceira(double valor)
    {
    }
}

public static class MovimentacaoFinanceiraFactory
{
    public static IMovimentacaoFinanceira CriarObjeto(TipoMovimentacaoFinanceira tipo)
    {
        switch (tipo)
        {
            case TipoMovimentacaoFinanceira.CONTA_PAGAR:
                return new ContaPagar();
            case TipoMovimentacaoFinanceira.CONTA_RECEBER:
                return new ContaReceber();
            case TipoMovimentacaoFinanceira.COBRANCA:
                return new Cobranca();
            default:
                throw new Exception("Tipo de operação financeira não implementado");
        }
    }
}

public class Financeiro
{
    private void GerarMovimentacaoFinanceira(double valor, TipoMovimentacaoFinanceira
tipo)
    {

MovimentacaoFinanceiraFactory.CriarObjeto(tipo).GerarMovimentacaoFinanceira(valor);
    }
}
```

Figura 25. Exemplo variações protegidas.

Percebe-se que foi delegada a uma *factory* a responsabilidade de definir e criar o objeto de movimentação financeira, um padrão comum utilizado em projetos. Também se criou uma interface IMovimentacaoFinanceira com o método GerarMovimentacaoFinanceira, assim todo tipo de operação financeira deve implementar esta interface.

Com estes ajustes, está encapsulado na *factory* a variação de código existente, e sempre que algum objeto precisar gerar uma movimentação financeira este solicitará à esta *factory* para resolver o tipo correto. Agora, quando for necessário adicionar mais um tipo de movimentação financeira, o único ponto do sistema que necessita de alteração é o *switch* dentro da *factory*, e este é o ponto chave destes ajustes.

2.3. Favorecer composição sobre herança (*favor composition over inheritance*)

Em linguagens de programação, o termo composição é utilizado quando um objeto possui uma referência interna a outro objeto, podendo ser por intermédio de campos, membros, propriedades ou atributos. Já o termo herança é utilizado quando uma classe A é criada estendendo uma classe B, tornando-se assim uma subclasse de B e B tornando-se assim uma superclasse. Uma classe A deve herdar de uma classe B quando pode-se dizer que A é um B. Esta técnica normalmente é utilizada quando há intenção de reaproveitar código ou comportamentos da superclasse, permitindo também os sobrescrever com o intuito de especializá-los.

O princípio *favor composition over inheritance* diz para favorecer a utilização de composição sobre herança, uma vez que ao utilizar composição ganha-se flexibilidade e a possibilidade de alterar o comportamento de uma classe em tempo de execução, diferentemente da herança onde o comportamento é definido em tempo de compilação.

Para exemplificar este princípio, pode-se ter como exemplo algo que lembra o exemplo dos pássaros utilizado no princípio de segregação de interface do SOLID. Suponha-se que o cliente solicita que seja desenvolvido um programa relacionado a pássaros e que este programa precisa fazer com que os pássaros cantem, voem e sejam exibidos. Logo, pode-se pensar que como cantar e voar são comportamentos comuns em pássaros pode-se então criar uma classe genérica que representa um pássaro e atribuir a ela os métodos necessários a fim de reaproveitar o código para todos os pássaros, deixando apenas um método abstrato responsável pela exibição do pássaro e que será especializado em cada tipo de pássaro:

```
public abstract class Passaro
{
    public virtual void Cantar()
    {
        // Faz o pássaro cantar
    }

    public virtual void Voar()
    {
        // Faz o pássaro voar
    }

    public abstract void Exibir();
}
```

Figura 26. Exemplo de violação variações protegidas.

O problema com esta abordagem é que existem pássaros que não voam e nem cantam, logo, esta arquitetura não consegue suprir a necessidade do programa, sendo necessário tratar estas exceções. Outra abordagem seria remover estes comportamentos da classe base e separá-los em interfaces, criando assim as interfaces IPassaroCantor, IPassaroOviparo e IPassaroVoador para que então cada pássaro implemente apenas as interfaces necessárias. Porém, ela contém um grave problema: se for necessário alterar a forma como os pássaros botam ovos, cantam ou voam, também é necessário alterar todos os métodos já implementados de cada classe já criada, aumentando assim a possibilidade de erros, sem falar na alta carga de retrabalho, além de alterar código já testado e que talvez já esteja em produção.

Para criar comportamentos ou alterar comportamentos já existentes sem reescrever o código já existente no sistema pode-se primeiramente recorrer ao princípio de variações protegidas do GRASP: identificar os trechos de código que variam e separá-los dos trechos que permanecem inalterados. E uma forma de aplicar este princípio neste exemplo é favorecer a composição sobre herança. Neste caso, pode-se identificar que os trechos de código que variam e necessitam serem sobrescritos são os métodos BotarOvo e Voar. Pode-se também observar que há comportamentos variados nestes métodos, onde há os pássaros que botam ovos, os que não botam ovos, os que voam e os que não voam. Partindo-se destas informações pode-se então pensar na criação de duas interfaces, uma para o comportamento de botar ovo e uma para o comportamento de voo:

```
public interface IComportamentoDeBotarOvo
{
    void BotarOvo();
}

public interface IComportamentoDeVoo
{
    void Voar();
}
```

Figura 27. Exemplo variações protegidas.

E a partir delas definir os diversos tipos de comportamentos identificados:

```
public class ComportamentoDePassaroQueBotaOvo : IComportamentoDeBotarOvo
{
    public void BotarOvo()
    {
        // Faz o pássaro botar um ovo
    }
}

public class ComportamentoDePassaroQueNaoBotaOvo : IComportamentoDeBotarOvo
{
    public void BotarOvo()
    {
        // Não faz nada pois o pássaro não bota ovo
    }
}

public class ComportamentoDePassaroQueVoa : IComportamentoDeVoo
{
    public void Voar()
    {
        // Faz o pássaro voar
    }
}
```

```
public class ComportamentoDePassaroQueNaoVoa : IComportamentoDeVoo
{
    public void Voar()
    {
        // Não faz nada pois o pássaro não voa
    }
}
```

Figura 28. Exemplo variações protegidas.

E por fim reestruturar a classe Passaro para permitir trabalhar com estes comportamentos criados:

```
public abstract class Passaro
{
    private IComportamentoDeBotarOvo comportamentoDeBotarOvo;
    private IComportamentoDeVoo comportamentoDeVoo;

    public void DefinirComportamentoDeBotarOvo(IComportamentoDeBotarOvo
comportamentoDeBotarOvo)
    {
        this.comportamentoDeBotarOvo = comportamentoDeBotarOvo;
    }

    public void DefinirComportamentoDeVoo(IComportamentoDeVoo comportamentoDeVoo)
    {
        this.comportamentoDeVoo = comportamentoDeVoo;
    }

    public void BotarOvo()
    {
        // Dispara comportamento de botar ovo
        comportamentoDeBotarOvo.BotarOvo();
    }

    public void Cantar()
    {
        // Faz o pássaro cantar
    }

    public void Voar()
    {
        // Dispara comportamento de voar
        comportamentoDeVoo.Voar();
    }

    public abstract void Exibir();
}
```

Figura 29. Exemplo variações protegidas.

Aqui os comportamentos referentes a botar ovo e voar na classe Passaro foram encapsulados. Agora estes comportamentos são definidos com o uso de métodos em tempo de execução e não mais em tempo de compilação como ocorre na herança, podendo inclusive serem alterados a qualquer momento durante a execução do programa, ou seja, ganha-se flexibilidade. Agora com estas alterações, pode-se criar as classes relacionadas ao beija-flor, kiwi e avestruz da seguinte forma:

```
public class BeijaFlor : Passaro
{
    public BeijaFlor()
    {
        var comportamentoDePassaroQueBotaOvo = new ComportamentoDePassaroQueBotaOvo();
        var comportamentoDePassaroQueVoa = new ComportamentoDePassaroQueVoa();

        DefinirComportamentoDeBotarOvo(comportamentoDePassaroQueBotaOvo);
        DefinirComportamentoDeVoo(comportamentoDePassaroQueVoa);
    }

    public override void Exibir()
    {
        // Exibe a imagem de um beija-flor
    }
}

public class Kiwi : Passaro
{
    public Kiwi()
    {
        var comportamentoDePassaroQueBotaOvo = new ComportamentoDePassaroQueBotaOvo();
        var comportamentoDePassaroQueNaoVoa = new ComportamentoDePassaroQueNaoVoa();

        DefinirComportamentoDeBotarOvo(comportamentoDePassaroQueBotaOvo);
        DefinirComportamentoDeVoo(comportamentoDePassaroQueNaoVoa);
    }

    public override void Exibir()
    {
        // Exibe a imagem de um kiwi
    }
}

public class CucoRelogio : Passaro
{
    public CucoRelogio()
    {
        var comportamentoDePassaroQueNaoBotaOvo = new
ComportamentoDePassaroQueNaoBotaOvo();
        var comportamentoDePassaroQueNaoVoa = new ComportamentoDePassaroQueNaoVoa();

        DefinirComportamentoDeBotarOvo(comportamentoDePassaroQueNaoBotaOvo);
        DefinirComportamentoDeVoo(comportamentoDePassaroQueNaoVoa);
    }

    public override void Exibir()
    {
        // Exibe a imagem de um relógio tipo cuco
    }
}
```

Figura 30. Exemplo variações protegidas.

Por intermédio da composição ganhou-se flexibilidade no código, além de permitir alterar o comportamento de objetos em tempo de execução. Se for necessária a criação de mais um comportamento de voo, por exemplo, basta criar uma classe que implementa a respectiva interface IComportamentoDeVoo, sem a necessidade de alterar o código já escrito.

3. Trabalhos relacionados

O trabalho de Souza (2018) apresentado ao programa de mestrado em engenharia de software do centro de estudos e sistemas avançados do recife faz uma análise sobre o impacto de *code smells* nas mudanças de software com o objetivo de "entender se a presença desses *code smells* tem alguma relação com as modificações feitas ao longo da manutenção e evolução dos produtos de software" (SOUZA, 2018, p.7). O *code smells* é um termo utilizado quando há problemas no código como código duplicado, métodos muito longos, classes extensas, problemas estruturais em geral, de reutilização, de extensão, etc.

Neste trabalho, são analisados dezessete *releases* de dois projetos de software diferentes e os resultados mostram que "classes que possuem *code smells* sofrem mais mudanças do que as demais classes. Além disso, *Large Class* é o *code smell* que tem mais impacto nessas mudanças dentre todos os analisados" (SOUZA, 2018, p.50). *Large Class* significa classe grande ou extensa.

O problema com *code smells* está diretamente ligado às alterações que os componentes de um sistema de *software* sofrem ao longo da vida útil do sistema, principalmente durante o seu período de manutenção e evolução. E são estes problemas que os princípios de design apresentados nesta pesquisa tentar resolver. Logo, pode-se perceber que há uma forte relação entre estas duas pesquisas, uma vez que os problemas de *code smells* podem ser reduzidos com o conhecimento destes princípios.

O trabalho de Torres (2016) apresentado ao programa de pós-graduação em ciência da computação (PROCC) da Universidade Federal de Sergipe (UFS) faz uma identificação e análise de clones de códigos heterogêneos em um ambiente corporativo de desenvolvimento de software. Ele comenta que "a exigência por acelerar o desenvolvimento de software nas empresas desencadeia uma série de problemas relacionados à organização do código" e que estes clones tornam "o aprimoramento e manutenção dos sistemas cada vez mais dificultado" (TORRES, 2016, p.4). O autor ainda comenta que "o desenvolvimento de software aliada à ausência de padrões e à inexistência de políticas internas que implementam melhores práticas, desencadeiam uma série de problemas relacionados à organização do código" (TORRES, 2016, p.7).

Este estudo realiza um experimento com dados de repositórios *open* e *closed-source* e conclui que "a incidência de clones não estava diretamente ligada ao número de linhas de código que os sistemas possuíam" e que "sistemas procedurais apresentaram menos incidência de clones que os orientados a objetos, reforçando a ideia de que linguagens com característica OO não necessariamente devem ser mais eficientes que as procedurais, no tratante ao surgimento de clones" (TORRES, 2016, p.53). OO é uma sigla para linguagem orientada a objetos. Esta conclusão reforça a afirmação de que o conhecimento sobre princípios de design em linguagens de programação orientadas a objetos é fundamental para o desenvolvimento de um sistema com paradigma OO, uma vez que a duplicação de código é um dos problemas que estes princípios tentam resolver, aprimorando a reutilização e organização do código.

Na monografia de Silva (2016), apresentada para obtenção de grau de bacharel em ciência da computação das Faculdades Integradas de Caratinga, comenta-se que um dos fatores que interferem na baixa utilização de padrões de projeto de software é devido a essa atividade não ser uma tarefa simples (SILVA, 2016, p.6) e que "muitos

desenvolvedores pensam ser perda de tempo e uma queda na produtividade por causa do tempo gasto para padronização do código-fonte" (SILVA, 2016, p.6). Além disso, ele também comenta que "a falta de conhecimento sobre os padrões faz com que muito tempo seja investido na manutenção do software por muitas das vezes não se ter uma maleabilidade" (SILVA, 2016, p.6). Silva faz uma abordagem sobre o perfil profissional dos desenvolvedores das empresas de software em geral, que por desconhecerem padrões de projeto, acabam criando soluções difíceis de entender, com baixa reutilização, manutenção desfavorável, baixa flexibilidade e pouca ou nenhuma possibilidade de expansão.

Silva conclui que a utilização de padrões de projeto gera "uma grande contribuição para diminuição no tempo de manutenção e respectivamente evolução de web service RESTful, pois torna-o mais flexível e apto a mudanças" (SILVA, 2016, p.60) e que "a não utilização de padrões de projeto está diretamente ligada a falta de conhecimento aprofundado sobre padrões de projeto de software" (SILVA, 2016, p.60). Embora a monografia de Silva tenha enfoque nos vinte e três padrões apresentados por Erich Gamma (GAMMA, 2000), estes padrões tentam solucionar problemas similares aos apresentados nos princípios GRASP e SOLID, melhorando a arquitetura e organização dos sistemas de *software*.

Em comparação a estes trabalhos, esta pesquisa tem foco em evitar que os problemas levantados por eles ocorram em um projeto de *software* por intermédio do estudo e da aplicação de princípios de manutenção e a utilização de algum mecanismo para tornar estas práticas um hábito na equipe. Os *code smells* mencionados por Souza, os códigos heterogêneos levantados por Torres e os problemas de manutenção e perfil profissional dos desenvolvedores mencionados por Silva são problemas que esta pesquisa tenta sanar com a disseminação destes princípios dentro da equipe e a utilização do *checklist* de boas práticas. Muitos dos problemas de manutenção levantados por estes trabalhos também são mencionados nesta pesquisa quando os princípios não são utilizados.

4. Metodologia da pesquisa

Nessa seção são descritos aspectos relacionados a metodologia da pesquisa como o tipo de enfoque, paradigma, os métodos escolhidos, a coleta e análise de dados, a população-alvo e as etapas da pesquisa.

4.1. Delineamento da pesquisa

Esta pesquisa possui caráter exploratório pois, segundo GIL (2008, p. 27), as pesquisas exploratórias "têm como principal finalidade desenvolver, esclarecer e modificar conceitos e ideias, tendo em vista a formulação de problemas mais precisos ou hipóteses pesquisáveis para estudos posteriores". Nela são aprofundadas informações sobre importantes princípios de design para linguagens de programação orientadas a objetos voltados a manutenção de sistemas de *software* com o intuito de repassar estas informações por intermédio de um *workshop* a uma equipe de programadores.

Esta pesquisa também possui um paradigma com enfoque qualitativo (fenomenológico). O enfoque qualitativo "procura o que é comum, mas permanece aberto para perceber a individualidade e os significados múltiplos, deixando de enfatizar uma necessidade de buscar uma média estatística" (ROESCH, 1999). Com o uso da

percepção, é traduzido em números opiniões e informações coletadas em um questionário para classificá-la e analisá-la.

4.2. Método de pesquisa

As estratégias utilizadas no processo de investigação dessa pesquisa são a pesquisa bibliográfica e o estudo de caso.

Segundo LAKATOS e MARCONI (2009), a pesquisa bibliográfica abrange todo o referencial teórico já tornado público em relação ao tema de estudo, como publicações avulsas, boletins, jornais, revistas, livros, pesquisas, monografias, teses, material cartográfico, meios de comunicação orais e audiovisuais. “A pesquisa bibliográfica não é mera repetição do que já foi dito ou escrito sobre certo assunto, mas propicia o exame de um tema sob novo enfoque ou abordagem, chegando a conclusões inovadoras”. LAKATOS e MARCONI (2009, p.57). A pesquisa bibliográfica permite a construção de um referencial teórico a respeito de existentes princípios de design para linguagens de programação orientadas a objetos, como SOLID e GRASP, voltados a manutenção de sistemas de *software*. Este referencial também possibilita a análise do código-fonte do sistema da unidade de análise.

Um estudo de caso “é uma investigação empírica que investiga um fenômeno contemporâneo dentro de seu contexto da vida real, especialmente quando os limites entre o fenômeno e o contexto não estão claramente definidos” (YIN, 2010, p. 39). Esta investigação empírica tem como seu componente trechos do ambiente de produção de um sistema Web de uma empresa real.

4.3. Unidade de análise e sujeitos participantes

O escopo desta pesquisa envolve o código de um sistema de gestão empresarial (ERP) *Web* voltado a gráficas digitais. Este sistema pertence a empresa Holdprint Sistemas Inteligentes, localizada na cidade de Novo Hamburgo, RS. Também faz parte deste escopo a equipe de programadores deste ERP, os sujeitos participantes desta pesquisa. Por questões de confidencialidade, seus nomes verdadeiros nomes não são revelados:

Tabela 1. Anos de experiência em desenvolvimento dos sujeitos participantes

IDENTIFICAÇÃO	EXPERIÊNCIA (ANOS)
Dev A	9
Dev E	11
Dev G	13
Dev J	12
Dev L	9
Dev R	-
Dev T	10

Todos os participantes são programadores com uma boa experiência no desenvolvimento de sistemas de *software*, mas desconhecem ou conhecem superficialmente os princípios aprofundados nesta pesquisa. Apenas um dos programadores acaba por não querer revelar sua experiência profissional.

4.4. Coleta de dados

Com a pesquisa bibliográfica e o aprofundamento dos princípios de design para linguagens de programação orientadas a objetos voltados a manutenção de sistemas de *software* realizados, a coleta de dados consiste em:

- Uma análise documental do código-fonte do sistema da unidade de análise.
- A elaboração de um *checklist* (Apêndice A) para servir de auxílio à equipe de desenvolvimento durante a construção e refatoração de código.
- A apresentação de um *workshop* (Apêndice B) para os programadores da equipe alvo, onde são apresentados todos os princípios de design e técnicas estudadas no desenvolvimento desta pesquisa.
- A realização de um questionário (Apêndice C) criado para captar a percepção dos participantes quanto ao material lhes apresentado.

4.5. Análise de dados

A partir dos dados qualitativos, coletados nas respostas do questionário e análise do código-fonte, é realizada uma análise de conteúdo.

Segundo LAKATOS e MARCONI (2009, p. 86), questionário "é um instrumento de coleta de dados constituído por uma série ordenada de perguntas, que devem ser respondidas por escrito e sem a presença do entrevistador". Neste questionário há questões abertas e questões fechadas de múltipla escolha.

Para a análise de código-fonte, é analisado e selecionado uma ou mais partes do código do sistema de *software* da unidade de pesquisa contendo problemas de manutenção. Estes problemas são detalhados levando em consideração os princípios de design desta pesquisa, e então sugestões de melhoria são realizadas.

4.6. Etapas da pesquisa

As etapas da pesquisa estão ilustradas na figura 31 desta seção.

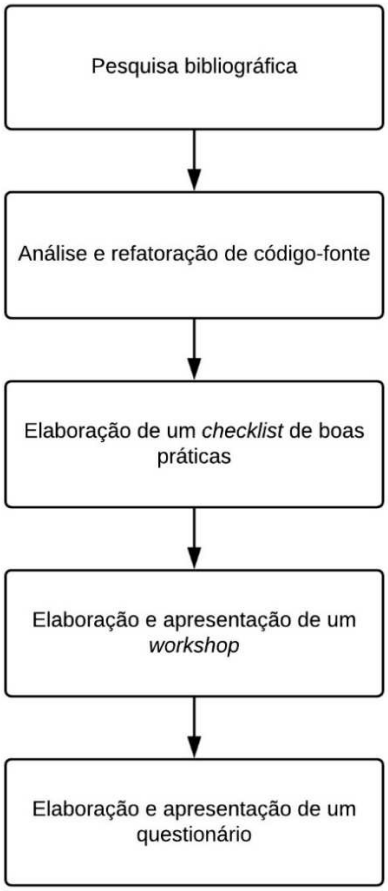

Figura 31. Fluxograma com as etapas da pesquisa.

Primeiramente, este estudo foi composto pela realização de uma pesquisa bibliográfica realizada para aprofundar o referencial teórico sobre princípios de design voltados a manutenção de sistemas de software. Estes princípios basearam-se principalmente nos princípios SOLID e GRASP, mas também houve menção sobre algumas práticas como favorecer composição sobre herança que ajudam a tornar os sistemas mais flexíveis a manutenções.

Concluída a etapa da pesquisa bibliográfica, deu-se início a análise e refatoração de códigos de produção do sistema alvo onde os princípios voltados a manutenção poderam ser aplicados. Esta refatoração serviu como exemplo em um ambiente real de desenvolvimento de como códigos já implementados pelos programadores da empresa alvo podem ser aprimorados utilizando estas técnicas, e com isto também se aprimorou a absorção de conhecimento pelos programadores já que os poderam ver exemplos familiares.

Após a análise e rafatoração de códigos de produção, houve a elaboração de um *checklist* de boas práticas de codificação (Apêndice A) focadas nos princípios estudados neste trabalho. O intuito foi que este *checklist* torne-se um hábito e um auxiliador durante o desenvolvimento e refatoração de códigos e que seja utilizado como uma das etapas do *code review* dos projetos, ajudando a aprimorar o código da empresa alvo no quesito manutenção.

Com estas etapas anteriores concluídas, deu-se início a elaboração e apresentação de um *workshop* (Apêndice B) para os programadores da equipe alvo com a finalidade de transmitir todo o conhecimento adquirido neste estudo, explicando os princípios estudados, apresentando as refatorações de código e o *checklist* de boas práticas como um recurso auxiliador.

Por fim, foi realizada a elaboração e apresentação de um questionário (Apêndice C) para os programadores da equipe alvo que teve o propósito de coletar *feedbacks* sobre todos os itens apresentados.

5. Desenvolvimento e análise dos resultados

Nesta seção é apresentada a análise sobre os resultados da pesquisa sobre princípios de design estudados, a análise dos códigos selecionados para refatoração e os resultados da apresentação deste trabalho a equipe de programadores alvo deste estudo.

5.1. Análise e rafatoração de código-fonte

O intuito da refatoração de código é analisar códigos passíveis de refatoração e refatorá-los aplicando os princípios desenvolvidos nesta pesquisa. Esta refatoração é apresentada à equipe de programadores da unidade de análise, para que possam examinar de forma prática a aplicação destes princípios em seus próprios códigos.

O código-fonte analisado pertence a um projeto .NET do tipo WebAPI escrito na linguagem C#. O código analisado está em ambiente de produção e corresponde a última versão do sistema *Web* de uma empresa real. A análise é embasada nos princípios estudados no referencial teórico deste artigo e tem um foco maior na manutenção do sistema. Ressalva-se que há a seleção de trechos de código chave como exemplos de violação dos princípios estudados, e estes trechos selecionados servem como exemplo para correções no restante do sistema. Estes exemplos de violações e suas correções são apresentadas no *workshop* para a equipe de desenvolvedores.

Alguns princípios do estudo não são citados nesta refatoração pois não se encontram evidências de códigos em produção que os ferem.

5.1.1. Princípios da responsabilidade única, baixa coesão e alto acoplamento

Há uma classe chamada *ProductService* e que possui mais de duas mil linhas de código. Isto se deve ao fato desta classe possuir diversas responsabilidades. Com o passar do tempo, foram-se adicionando diversas outras responsabilidades a ela além da responsabilidade inicialmente proposta, que é o tratamento de um produto no sistema.

O princípio da responsabilidade única (SOLID) é ferido, uma vez que a classe *ProductService* possui diversas responsabilidades, o que dificulta sua manutenção e aumenta sua fragilidade, uma vez que com várias responsabilidades entrelaçadas em uma única classe, fica mais difícil alterar uma destas responsabilidades sem comprometer outras. Esta classe também possui uma baixa coesão (GRASP), uma vez que suas diversas responsabilidades e seu código extenso tornam-na difícil de entender, reusar e manter. Além, disto esta classe possui um alto acoplamento (GRASP), pois depende de muitas outras classes, e mudanças relacionadas a estas classes podem resultar em mudanças à própria classe.

Como proposta de solução, deve-se identificar, organizar e separar desta classe estas responsabilidades extras, tornando esta classe e as demais classes criadas mais coesas e de fácil reuso, além de corrigir todos os problemas mencionados, tornando a manutenção mais fácil. Um exemplo destas responsabilidades extras está no método *GenerateContext* (Anexo A). Este método é responsável por criar um contexto de informações a partir dos produtos selecionados pelo usuário durante o orçamento. Para suportar esta função, existem diversos outros métodos que acabam poluindo a classe *ProductService*. Logo, propõe-se como etapa inicial a criação de uma classe chamada *ProductContextGenerator* e uma interface chamada *IProductContextGenerator* que agregarão todos estes métodos de suporte para a geração de contexto. Este processo de identificação, organização e separação destas responsabilidades em demais classes deve ser seguido até que as classes se tornem coesas e com isto a manutenção é muito facilitada:

```
public interface IProductContextGenerator
{
    void GenerateContext(BudgetProduct budgetProduct);
}

public class ProductContextGenerator : IProductContextGenerator
{
    public void GenerateContext(BudgetProduct budgetProduct)
    {
        // Geração do contexto do produto
    }

    // Demais métodos que dão suporte à geração do contexto
}
```

Figura 32. Exemplo refatoração SRP, alta coesão e baixo acoplamento.

5.1.2. Princípio de aberto-fechado

Em uma classe chamada *FieldService*, há um método chamado *GetByType* (Anexo B). Este método cria uma instância de tipo de classe de conteúdo baseando-se no parâmetro *FieldContentType* que é um enumerador.

O princípio de aberto-fechado (OCP) é ferido pois caso um novo valor seja adicionado no enumerador *FieldContentType*, é necessário varrer todo o sistema e modificar todos os trechos relacionados. Caso algum ponto seja esquecido, um novo erro é criado no sistema, o que possivelmente é o caso deste *swtich*, pois observando-se o enumerador, percebe-se que há muitos valores que não estão neste *switch*. Além disto, caso estes pontos estejam em outras bibliotecas externas, estas precisam ser recompiladas e redistribuídas, aumentando mais ainda as chances de criação de novos erros. Além do mais, estes trechos já foram testados e já estão em produção. Quanto maior a quantidade de código alterada, maior a possibilidade de surgimento de novos erros.

Como proposta de solução, propõe-se a criação de uma *factory*: é um padrão de design onde uma classe é responsável pela criação de objetos complexos. Ela pode ser chamada de *HoldprintContentFactory* e possuir um método semelhante ao visto acima, onde com o uso do *switch* ela define qual classe concreta é instanciada de acordo com o enumerador passado como parâmetro. Desta forma, delega-se a responsabilidade de criar estes objetos a uma classe especializada e mantem-se o *switch* em apenas um local,

para que quando for criada mais uma condição seja modificado apenas um local, mantendo o sistema fechado para modificação e aberto a extensão:

```
public enum FieldContentType
{
    Texto = 1,
    AutoComplete = 2,
    LongText = 3,
    Numbers = 4,
    Currency = 5,
    CheckboxOptions = 6,
    RadioOptions = 7,
    User = 8,
    Enterprise = 9,
    Person = 10,
    Phone = 11,
    Time = 12,
    IntervalTime = 13,
    Date = 14,
    IntervalDate = 15,
    Address = 16,
    SwitchButton = 17,
    FeedstockFilter = 19,
    InkSelector = 20,
    EquipmentsOptions = 21,
    InkTypeSelector = 22,
    SupplierList = 23,
    BillingConfig = 24,
    PriceConfig = 25
}

public interface IHoldprintContentFactory
{
    FieldContent GetByType(FieldContentType fieldContentType);
}
```

```
public class HoldprintContentFactory
{
    public FieldContent GetByType(FieldContentType fieldContentType)
    {
        switch (fieldContentType)
        {
            case FieldContentType.AutoComplete:
            case FieldContentType.CheckboxOptions:
                return new HoldprintArrayContent(new List<string>());
            case FieldContentType.Time:
                return new HoldprintTimeContent(string.Empty);
            case FieldContentType.IntervalTime:
                return new HoldprintIntervalTimeContent(string.Empty);
            case FieldContentType.Date:
                return new HoldprintDateContent();
            case FieldContentType.IntervalDate:
                return new HoldprintDateIntervalContent();
            case FieldContentType.FeedstockFilter:
                return new HoldprintFeedstockFilterContent();
            case FieldContentType.InkTypeSelector:
                return new HoldprintNumberContent(0);
            case FieldContentType.SwitchButton:
                return new HoldprintBooleanContent(false);
            default:
                return new HoldprintTextContent(string.Empty);
        }
    }
}

public class FieldService
{
    private readonly IHoldprintContentFactory holdprintContentFactory;

    public FieldService(IHoldprintContentFactory holdprintContentFactory)
    {
        // Princípio da inversão de dependência: depender de uma abstração
        // e não de uma classe concreta.
        this.holdprintContentFactory = holdprintContentFactory;
    }

    private FieldContent GetByType(FieldContentType fieldContentType)
    {
        // Delega a criação do objeto FieldContent a uma classe especializada,
        // aprimorando a manutenção com o reuso e mantendo a lógica de criação em
        // um único local.
        return holdprintContentFactory.GetByType(fieldContentType);
    }
}
```

Figura 33. Exemplo refatoração aberto-fechado.

5.1.3. Princípio da segregação de interface

Há uma interface chamada *IProcessService* (Anexo C) que possui trinta e sete métodos a serem implementados. O princípio da segregação de interface (ISP) diz que clientes não devem ser forçados a implementar interfaces que não usam ou não devem ser forçados a depender de métodos que não usam. É recomendado utilizar interfaces específicas e mais especializadas do que uma interface genérica. Caso algum cliente precise implementar apenas uma parte da interface ele será obrigado a implementar

todos os trinta e sete métodos, mesmo sem necessidade. Desta forma, recomenda-se que esta interface, e várias outras espalhadas pelo sistema, sejam divididas em interfaces menores com famílias de métodos coesos, para que então, caso um cliente precise implementá-los, este possa implementar somente o necessário.

Propõe-se iniciar esta divisão definindo-se qual o objeto mais coeso para a interface. Para a interface *IProcessService* o objeto mais coeso é o *Process*, que define um processo no sistema. Com esta definição, pode-se agora dividir esta interface em várias interfaces coesas, separando dela métodos que trabalham com outros objetos, que neste caso são *ProcessCostEngineeringModel*, *GraphQLQuery*, *IHoldprintFilter*, *ProcessFamily*, *RelatedProcess, BudgetProcess* e *UserPreference*. Em seguida, pode-se pensar em aplicar o padrão *Command Query Responsibility Segregation* (CQRS), onde basicamente é criado um modelo para atualizar dados (comandos ou *commands*) e outro modelo para ler dados (consultas ou *queries*), podendo-se dividir a interface *IProcessService* em duas, *IProcessServiceCommands* e *IProcessServiceQueries*:

```
public interface IProcessServiceCommands
{
    Process AddChecklist(Checklist checklist, string id);
    Process AddCustomField(Field field, string id);
    Process AddWarning(Warning warning, string id);
    Process Create(Process process);
    Process Delete(int publicId);
    Process Patch(JsonPatchDocument<Process> patchItem, int publicId, string id);
    Process Update(Process process);
}

public interface IProcessServiceQueries
{
    Process DiscoveryChecklist(Process process);
    Process Find(string id);
    Process Find(int publicId);
    Process Find(int publicId, int version);
    List<Process> FindByMecId(int mecPublicId);
    List<Process> FindProcessesFromList(string[] ids);
    List<Process> FindProcessesFromPublicIds(string @params);
    List<Process> FindProcessesFromPublicIds(List<int> publicIds);
    List<Process> List();
}
```

Figura 34. Exemplo refatoração segregação de interface.

5.1.4. Princípio da inversão de dependência

Observa-se que a classe chamada *CitiesFactory* (Anexo D) possui uma forte dependência da classe concreta *ApplicationContext*, e isto se repete em várias partes do sistema, como no controlador *CitiesController* (Anexo E). O princípio da inversão de dependência (DIP) diz que módulos devem depender de abstrações e não de classes concretas e que módulos de alto nível não devem depender de módulos de baixo nível e que ambos devem depender de abstrações. Neste caso, pode-se considerar este controlador e esta classe como módulos de alto nível pois utilizam módulos de menor nível para exercerem suas funções. Caso este princípio não seja observado, o resultado é uma arquitetura frágil a mudanças, o que a torna cada vez mais difícil de ser mantida com a evolução do sistema.

O problema com esta arquitetura é que ela é altamente dependente de classes concretas, e caso ocorra alguma mudança em algum dos módulos de baixo nível esta mudança também a afeta, sendo necessário uma refatoração em todas as partes do sistema onde estes módulos são utilizados. Por exemplo, caso o contexto da aplicação mude para outro contexto, é necessário substituir todas as referências da classe *ApplicationContext*. Para solucionar este problema, deve-se abstrair estas dependências: ao invés de depender das classes concretas *ApplicationContext* e *CitiesFactory*, deve-se depender de algo que manipule o contexto da aplicação e de algo que trate cidades, sem saber quem implementa estas funções. Neste caso, pode-se substituir *ApplicationContext* por uma interface *IContext* e *CitiesFactory* por uma interface *ICitiesFactory*. *ApplicationContext* e *CitiesFactory* passam então a implementar estas interfaces, respectivamente.

Com estas ações, resolve-se o problema de dependência no módulo de baixo nível *CitiesFactory*, pois agora ele passa a ter um construtor com referência para a interface *IContext*. Porém, mesmo quando o sistema depende de abstrações, em algum momento é necessário informar quais são as classes concretas que implementam estas abstrações, e neste caso o controlador *CitiesController* ainda teria uma referência a estas classes concretas. Uma solução para essa questão é a utilização de um mecanismo de injeção de dependência. Com este mecanismo, pode-se facilmente configurar qual classe concreta é injetada no lugar de uma abstração, permitindo que o sistema mantenha somente abstrações nas dependências de suas classes, incluindo *controllers*:

```
public class CitiesFactory : ICitiesFactory
{
    private readonly IContext context;

    public CitiesFactory(IContext context)
    {
        // Princípio da inversão de dependência: depender de uma abstração
        // e não de uma classe concreta.
        this.context = context;
    }

    public Task Create(City city)
    {
        context.Cities.Add(city);
        return context.SaveChangesAsync();
    }

    public Task<List<City>> Read()
    {
        return context.Cities.ToListAsync();
    }

    public Task<City> Read(int id)
    {
        return context.Cities.FindAsync(id);
    }

    public Task<City> GetCityByIbgeId(int idIbge)
    {
        return context.Cities.FirstOrDefaultAsync(x => x.SpecificCountryCode == idIbge);
    }
}
```

```
public class CitiesController : ApiController
{
    private readonly IContext context;

    public CitiesController(IContext context)
    {
        // Princípio da inversão de dependência: depender de uma abstração
        // e não de uma classe concreta.
        this.context = context;
    }

    public async Task<List<City>> GetCities()
    {
        var citiesFactory = new CitiesFactory(context);
        return await citiesFactory.Read();
    }
}
```

Figura 35. Exemplo refatoração inversão de dependência.

5.1.5. Princípios da fabricação pura, indireção e variações protegidas

O princípio de especialista da informação diz que se deve atribuir uma responsabilidade a uma classe quando esta possui as informações necessárias para executá-la. Porém, há casos onde ao seguir este princípio acaba-se por diminuir a coesão e aumentar o acoplamento, pois a classe passa a ter mais responsabilidades e a depender de mais objetos. O princípio da fabricação pura diz que, quando esta situação ocorre, se deve criar uma nova classe para tratar esta responsabilidade extra, tornando assim as classes mais coesas e menos acopladas. Com esta nova classe cria-se uma indireção entre a classe original e os objetos dos quais antes ela dependia, pois agora ela acaba dependendo de um único objeto especializado que contém estes objetos. Em outras situações, como no caso deste exemplo que é escolhido, ao criar esta nova classe especializada também se acaba por criar uma variação protegida, pois as variações nesta nova classe não afetam a classe original.

O exemplo escolhido e que compõe os três princípios mencionados é o da classe *BudgetCalcConsumer* (Anexo F). A responsabilidade desta classe é consumir um cálculo do orçamento, disparado por um evento externo que indica quando o orçamento foi calculado pelo *software* de inteligência artificial. Porém, percebe-se que além desta responsabilidade, esta classe também é responsável criar um *CalcSnippet* com o uso de diversos objetos e por criar um novo escopo com o mecanismo de injeção de dependência, neste caso o *Autofac*. Para que estas responsabilidades extras sejam implementadas ela acaba tendo que conhecer diversos objetos, além de precisar saber detalhes de como criar um *CalcSnippet* e de como criar um novo escopo. Ocorre então neste caso os problemas de baixa coesão e de alto acoplamento.

O princípio da fabricação pura aplica-se neste exemplo pois os dados necessários para criar o *CalcSnippet* e o novo escopo estão na classe *BudgetCalcConsumer*. Porém, é necessário criar classes extras para que esta mantenha-se coesa, ao mesmo tempo que se cria uma indireção entre esta classe e os diversos objetos extras.

Por fim, ao criar estas indireções também se está protegendo variações, pois se por acaso a forma de obter um *CalcSnippet* ou a forma de criar um novo escopo na

injeção de dependência mudarem, não é necessário mudar a classe *BudgetCalcConsumer*.

Como solução de refatoração, propõe-se que o resolvedor de dependência, aqui chamado de *DependencyResolver*, seja passado no construtor desta classe, e que este tenha um método que trata a criação de um novo escopo de injeção de dependência. Desta forma, a classe *BudgetCalcConsumer* apenas sabe que existe um resolvedor de dependência e que este sabe criar um novo escopo, sem saber ou depender de detalhes da implementação. Propõe-se também que o *CalcSnippet* seja passado diretamente por parâmetro, sendo assim outra classe responsável por criá-lo:

```
public interface IDependencyResolver : IDisposable
{
    IDependencyResolver BeginLifetimeScope<T>(params object[] parameters);

    T Resolve<T>();
}

public class BudgetCalcConsumer : IBudgetCalcConsumer
{
    private readonly IDependencyResolver dependencyResolver;

    public BudgetCalcConsumer(IDependencyResolver dependencyResolver)
    {
        this.dependencyResolver = dependencyResolver;
    }

    public void ConsumeCalcStatus(CalcSnippet calcSnippet)
    {
        using (IDependencyResolver dr =
dependencyResolver.BeginLifetimeScope<IAccountMongoApplicationContext>(calcSnippet.Accoun
tId))
        {
            dr.Resolve<ICalculationService>().UpdateCalc(calcSnippet);
        }
    }
}
```

Figura 36. Exemplo refatoração fabricação pura, indireção e variações protegidas.

5.2. Apresentação do *workshop*

O *workshop* possui a finalidade de apresentar os princípios de design voltados a manutenção de sistemas de *software* estudados nesta pesquisa aos programadores da unidade de análise. Neste *workshop*, é realizada uma apresentação de *slides* (Apêndice B) contendo a introdução e exemplos de cada um destes princípios. Neste *workshop* também é apresentado exemplos da refatoração de código realizada com o código de produção. Espera-se que esta apresentação instigue os programadores a tentar conhecer mais sobre boas práticas de codificação, e que sirva como um passo inicial para que isto se torne um hábito no dia a dia. Nesta apresentação também é introduzido o *checklist* de boas práticas, material que será usado durante o desenvolvimento e durante o *code review*.

Figura 37. Foto da equipe de desenvolvedores.

5.3. *Checklist* de boas práticas de codificação

O *checklist* de boas práticas (Apêndice A) é um guia criado para os programadores da unidade de análise que deve servir de apoio quando estes estão criando ou refatorando códigos. Propõe-se que este guia seja uma das etapas realizadas durante o *code review* dos sistemas e que também seja utilizado como referência pelos programadores durante a construção ou refatoração de códigos.

O *code review* é uma etapa do desenvolvimento que ocorre quando um programador entrega uma tarefa de código. Nele, o programador escolhe outros programadores para realizarem uma validação e um controle de qualidade do código entregue. Caso o código tenha ressalvas, este volta para o programador que o criou para serem realizados os ajustes solicitados. Este processo se repete até que o código seja aprovado por todos os programadores.

Com a criação e utilização deste *checklist*, espera-se que a cultura de boas práticas seja cada vez mais absorvida pelos programadores até que vire uma prática diária. Os programadores da equipe alvo podem também colaborar com este *checklist*, propondo novas práticas ou ajustando as práticas existentes.

Como modelo inicial, propõe-se que estas práticas contenham os ensinamentos dos princípios estudados no referencial teórico deste trabalho que compõem os princípios SOLID e GRASP. O fluxo com as etapas do *code review* é ilustrado na figura 38.

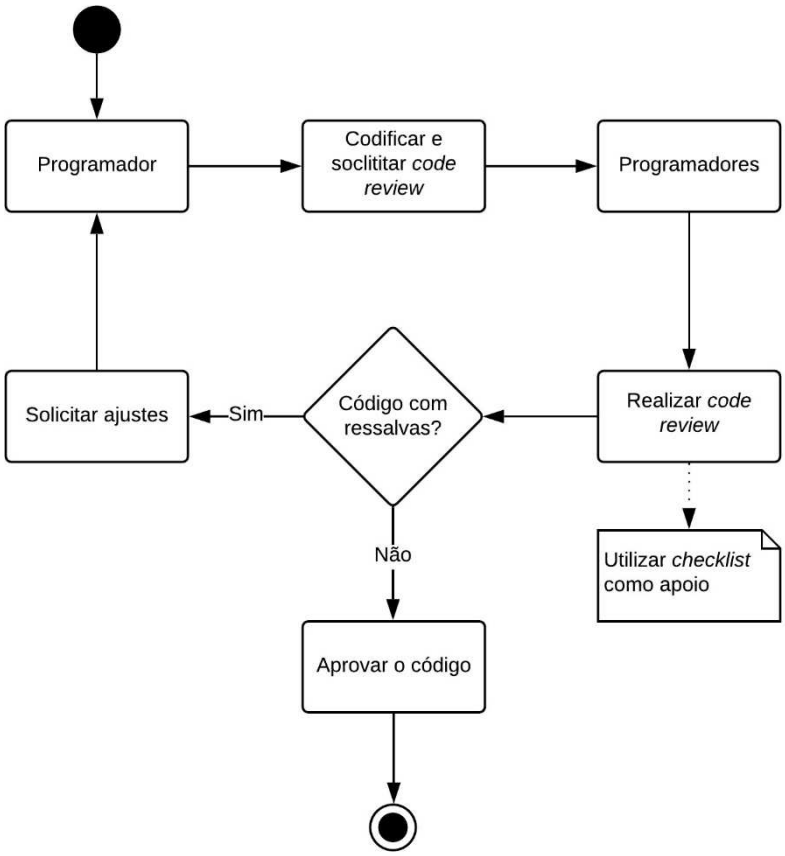

Figura 38. Etapas do *code review*.

5.4. Apresentação do questionário

O questionário (Apêndice C) é apresentado ao final do *workshop* com a finalidade de captar a percepção dos participantes quanto aos princípios apresentados. A finalidade do questionário é saber se de fato os participantes conseguiram compreender a importância de se conhecer e aplicar os princípios de codificação apresentados e se acharam que o material apresentado irá ajuda-los a codificar com melhor qualidade.

O questionário possui questões dissertativas, que servem para captar a percepção quanto ao trabalho apresentado e questões objetivas sobre os princípios SOLID e GRASP, onde os sujeitos participantes devem relacionar seus nomes com suas definições. A ideia das questões objetivas é fazer com que os desenvolvedores reflitam sobre os ensinamentos e tentem melhor absorvê-los.

5.5. Análise dos resultados

Com a pesquisa bibliográfica e o aprofundamento dos princípios de design para linguagens de programação orientadas a objetos voltados a manutenção de sistemas de *software* foi possível realizar uma análise documental do código-fonte do sistema da unidade de análise com foco em problemas de manutenção. Desta análise documental extraiu-se trechos de código do sistema passíveis de refatoração, onde estes foram refatorados aplicando-se os princípios estudados neste trabalho, e que serviram também

de exemplo aos programadores da equipe alvo, aprimorando seu entendimento quanto a estes princípios por estarem familiarizados com os códigos de exemplo.

O *checklist* de boas práticas (Apêndice A) foi possível graças ao estudo preliminar realizado durante o referencial teórico e já está rendendo frutos, pois está servindo de auxílio à equipe de desenvolvimento durante a construção e refatoração de código e também durante o processo de *code review*. A construção deste *checklist* foi de vital importância, pois com ele a equipe tem um apoio quanto a aplicação dos princípios e é uma forma de tornar a cultura de boas práticas cada vez mais natural à equipe.

A apresentação do *workshop* (Apêndice B) para os programadores da equipe alvo rendeu resultados excelentes e cumpriu seus objetivos. Com ele, foi possível apresentar à equipe todos os princípios de design e técnicas estudadas no desenvolvimento desta pesquisa. Em conversas posteriores, foi perceptível o grau de satisfação da equipe quanto ao conteúdo apresentado. Todos, sem exceção, gostaram muito dos princípios apresentados e também dos exemplos utilizados.

O questionário (Apêndice C) criado para captar a percepção dos participantes quanto ao material lhes apresentado serviu ao seu propósito. Ele conteve as seguintes perguntas e respostas dos participantes:

Tabela 2. Respostas da questão 1 do questionário.

Questão 1: Você acha que o workshop o ajudou a conhecer mais ou a compreender melhor sobre princípios de codificação voltados a manutenção de sistemas? Justifique sua resposta.
Programador 1: “Foi possível fazer uma relação nos código atuais do sistema em que estamos desenvolvendo e nos próximos que iremos desenvolver.”
Programador 2: “Sim. Com a apresentação juntamente aos exemplos foi possível compreender melhor os princípios a serem utilizados nos projetos..”
Programador 3: “Com certeza. Mesma para aqueles que já conheciam, é importante revisitar estes conceitos e ver sua aplicação sob o viés de outra pessoa.”
Programador 4: “Com certeza! Auxilia no desenvolvimento do sistema com técnicas de codificação simples e muito úteis, facilitando a manutenção do sistema devido ao código ficar mais claro, objetivo e coeso.”
Programador 5: “Sim. Não conhecia GRASP, e outros princípios não estavam mais claros na memória.”
Programador 6: “Sim, creio que ajudou muito. Principalmente sobre os exemplos práticos apresentados, normalmente quando se pesquisa conteúdo sobre boas práticas, carece muito de bons exemplos.”
Programador 7: “Sim. Além do conceito, a apresentação de exemplos práticos que utilizamos no dia a dia foi fundamental para o entendimento das boas práticas.”

Quando questionado aos programadores se acharam que o workshop os ajudou a conhecer mais ou a compreender melhor sobre princípios de codificação voltados a manutenção de sistemas todos foram enfáticos em responder que sim, e isto vai de encontro ao 4º objetivo específico estabelecido deste trabalho.

Os programadores 3 e 5 revelaram que já conheciam ou já ouviram falar de alguns dos princípios apresentados, mas que a apresentação serviu para relembrar ou reforçar a aplicação destes princípios.

Outra observação importante foi que os programadores 1, 2, 6 e 7 destacaram que os exemplos apresentados foram importantes para o entendimento destes princípios, e isto vai de encontro ao segundo objetivo específico estabelecido.

Tabela 3. Respostas da questão 2 do questionário.

Questão 2: Você acha que estes princípios o ajudarão a escrever códigos de melhor qualidade voltados a manutenção de sistemas? Justifique sua resposta.
Programador 1: “Ajudaram tanto na qualidade do código como no entendimento futuro a partir do próprio desenvolvedor.”
Programador 2: “Sim. E assim tornará o código bem mas fácil para realizar manutenção. “
Programador 3: “Certamente. A finalidade destes princípios justamente é escrever códigos com mais qualidade, e quanto melhor o time dominar estes conceitos, melhor será o resultado do trabalho da equipe.”
Programador 4: “Sim, conforme as tecnologiass evoluem, e isso ocorre com alta velocidade, há a necessidade de readequar o código em determinados momentos. Realizando a codificação aliado às boas praticas, a manuteção destes códigos se torna muito mais simplificada.”
Programador 5: “Sim. No momento estou criando um novo projeto e poderei aplicar este princípios deixando a aplicação com uma qualidade melhor.”
Programador 6: “Sim, pois quando for iniciar o desenvolvimento de algum recurso, irei utilizar o conteúdo apresentado para pensar em uma boa arquitetura antes de tudo.”
Programador 7: “Sim. Como a complexidade que a aplicação que desenvolvemos é grande ficou claro os pontos em que erramos e onde podemos melhorar.”

Quanto à questão se acham que estes princípios os ajudarão a escrever códigos de melhor qualidade voltados a manutenção de sistemas, novamente todas as respostas foram todas positivas. No geral, todos mencionaram que utilizarão estes conhecimentos no desenvolvimento de projetos atuais e futuros com o intuito de aprimorar o código da aplicação. O programador 7 mencionou também que o problema de manutenção se torna aparente em grandes aplicações, e que pôde com esta apresentação pensar com clareza nos pontos em que a equipe errou e onde ela pode melhorar.

Estas respostas se alinham ao objetivo principal desta pesquisa, que é aprimorar o conhecimento coletivo dos participantes quanto a princípios de codificação voltados a manutenção de sistemas para que então estes possam escrever códigos melhores em termos de manutenção.

Tabela 4. Respostas da questão 3 do questionário.

Questão 3: Você acha importante conhecer e aplicar boas práticas de codificação em nossos sistemas? Justifique sua resposta.
Programador 1: “Conhecendo o princípio e sabendo aplicá-lo de maneira correta quando necessário.”
Programador 2: “Sim. Qualquer prática que melhore o código é bastante válida.”
Programador 3: “Sim, sem dúvida. Aplicar boas práticas ajuda os profissionais a escreverem soluções de código mais coesas, visando baixo acoplamento de código, focando na qualidade das entregas, e diminuindo desta forma possíveis esforços necessários com retrabalho e correção de bugs.”
Programador 4: “Com toda a certeza! Como explicitado na resposta anterior, a prática deste conceito facilita tanto na manutenção quanto na elaboração de melhorias e acréscimos no código.”
Programador 5: “Sim. Os padrões são necessários para facilitar a manutenção de novos recursos e evitar bugs.”
Programador 6: “Tenho convicção que seja essencial, pois sem isso será repetido os mesmos

erros e dificuldade extrema em dar manutenção nos sistemas legados.”
Programador 7: “Sim. A aplicação de boas práticas é fundamental principalmente para a manutenção e vida útil do sistema levando em conta as mudanças que poderão ocorrer no decorrer do tempo.”

Ficou claro analisando as respostas da questão 3 que todos os programadores acham importante conhecer e aplicar boas práticas de codificação nos sistemas. Embora a resposta a esta pergunta pareça óbvia, o objetivo principal desta questão foi contrastar com o problema de pesquisa, que diz que embora a fase de manutenção de sistemas de software compreende a maior parte da sua vida útil (SOMMERVILLE, 2011), os programadores por muitas vezes não se preocupam em criar um código utilizando boas práticas e princípios, com o uso de padrões de projetos conhecidos e com foco na manutenção, e por estas razões acabam criando um código difícil de ser mantido.

Tabela 5. Respostas da questão 4 do questionário.

Questão 4: Sobre o checklist de boas práticas de codificação, você acha que ele pode contribuir com a melhora na qualidade do código-fonte da equipe? Justifique sua resposta.
Programador 1: “Quanto mais comum for ser identificado situações onde o checklist será consultado, mais o entendimento e a plicação dos conceitos serão concretizados.”
Programador 2: “Sim. Pelo fato de que passaremos a usar boas práticas, consequentemente irá melhorar a qualidade do código.”
Programador 3: “Pode ajudar sim, servindo como um guia de boas práticas na hora de implementar e de revisar algum código.”
Programador 4: “Conforme formos trabalhando as melhorias irão surgindo.”
Programador 5: “Sim. Pois estamos nivelando o nível de conhecimento da equipe para um patamar superior e definindo regras mais claras do que deve ser avaliado nas revisões.”
Programador 6: “Acredito que sim, pois sem o checklist facilmente poderá ser esquecido uma ou outra etapa.”
Programador 7: “Sim. Abordando alguns casos práticos que temos atualmente mostra que a utilização de boas práticas ajudam no melhor entendimento de regras de negócio e melhor definição do que cada classe do sistema é responsável por fazer.”

A questão 4 vai de encontro ao 3º objetivo específico deste estudo, que é elaborar um *checklist* de boas práticas de codificação para servir de auxílio aos programadores durante o desenvolvimento e para absorção gradual dos princípios estudados. Analisando as respostas desta pergunta, percebe-se que todos concordam que o *checklist* de boas práticas irá contribuir com a melhora na qualidade do código-fonte da equipe.

Este *checklist* foi desenvolvido com o propósito de ser utilizado no dia-a-dia pela equipe para que gradualmente ela vá absorvendo a cultura de boas práticas, até que isto se torne um hábito. Como muitos destes princípios não são fáceis de serem compreendidos e por muitas vezes são esquecidos, este *checklist* e sua utilização se tornam fundamental para este propósito.

Tabela 6. Respostas da questão 5 do questionário.

Questão 5: Além dos princípios apresentados hoje, você tem alguma sugestão de novas temáticas para futuros trabalhos ou workshops?
Programador 1: “desing patterns (padrões de projetos de software)”

Programador 2: “Não.”
Programador 3: “Um estudo muito interessante seria refatorar um sistema existente, legado talvez, que não utilize nenhum tipo de princípio de boas práticas. E após a refatoração do sistema, monitorar e analisar se houve uma melhora nos indicadores de abertura de bugs, retrabalho e manutenção de código.”
Programador 4: “No momento não.”
Programador 5: “Padrões para Algorítimo Genético.”
Programador 6: “Design patterns e DDD são duas coisas que acredito serem boas temáticas.”
Programador 7: “Design Patterns, padrões de projetos, micro servicos.. “

Espera-se que todo o trabalho apresentado a equipe seja apenas o início da utilização de boas práticas e que ele instigue a equipe a pensar em como aprimorar o desenvolvimento de seus sistemas, e isto não envolve somente estes princípios. Por esta razão a questão 5 foi elaborada.

Por estas respostas e pela percepção de conversas posteriores com a equipe, foi percebido que a maioria ficou empolgada com o que mais se possa fazer para aprimorar seus sistemas e que já estão pensando em futuros *workshops*. Logo, é possível dizer que o *feedback* da equipe foi excelente quanto ao *workshop* e que os objetivos da pesquisa foram alcançados.

Sumarizando as respostas das perguntas 1 a 4, removendo a pergunta 5 pois esta é uma sugestão, e dividindo estas perguntas entre respostas positivas e negativas temos um cenário com 100% de respostas positivas.

Além de questões dissertativas, que serviram para captar a percepção quanto ao trabalho apresentado, a pesquisa também apresentou questões objetivas sobre os princípios SOLID e GRASP, onde os sujeitos participantes deveriam relacionar seus nomes com suas definições. O objetivo destas questões não foi com que todos soubessem em uma única apresentação sobre todos os 14 princípios apresentados, mas sim que parassem para refletir sobre os ensinamentos e tentassem melhor absorvê-los. Após todos terminarem o questionário, foi lhes apresentado as respostas corretas para que este objetivo pudesse ser alcançado.

Tabela 7. Respostas da questão 6 do questionário.

Questão 6: Uma classe deve possuir somente uma única responsabilidade, podendo haver somente um único motivo pela qual ela mude.
Programador 1: “Princípio da responsabilidade única e alta coesão”
Programador 2: “Princípio da responsabilidade única e alta coesão”
Programador 3: “Princípio da responsabilidade única e alta coesão”
Programador 4: “Princípio da responsabilidade única e alta coesão”
Programador 5: “Princípio da responsabilidade única e alta coesão”
Programador 6: “Princípio da responsabilidade única e alta coesão”
Programador 7: “Princípio da responsabilidade única e alta coesão”

Talvez os princípios da responsabilidade única (SOLID) e alta coesão (GRASP), que são relacionados, sejam os dois princípios mais importantes entre todos os outros pois são fundamentais para um código de qualidade e tem um impacto profundo na manutenção de sistemas. Um código coeso é facilmente compreendido e mantido. Todas as respostas quanto a questão 6 estão corretas.

Tabela 8. Respostas da questão 7 do questionário.

Questão 7: Qualquer classe derivada deve ser substituível por sua classe base sem mudar o seu comportamento ou o funcionamento correto do programa.
Programador 1: “Princípio da inversão de dependência”
Programador 2: “Princípio da substituição de Liskov“
Programador 3: “Princípio da substituição de Liskov“
Programador 4: “Princípio da substituição de Liskov“
Programador 5: “Princípio da substituição de Liskov“
Programador 6: “Princípio da substituição de Liskov“
Programador 7: “Princípio da substituição de Liskov“

Apenas o programador 1 errou a resposta da questão 7, que é o Princípio da substituição de Liskov. Porém, após a revelação das respostas corretas ele comentou que se precipitou quanto a esta pergunta.

Tabela 9. Respostas da questão 8 do questionário.

Questão 8: Módulos de software (classes e métodos) devem ser abertos para extensão e fechados para modificação.
Programador 1: “Princípio aberto-fechado”
Programador 2: “Princípio aberto-fechado”
Programador 3: “Princípio aberto-fechado”
Programador 4: “Princípio aberto-fechado”
Programador 5: “Princípio aberto-fechado”
Programador 6: “Princípio aberto-fechado”
Programador 7: “Princípio aberto-fechado”

Todos responderam corretamente à questão 8.

Tabela 10. Respostas da questão 9 do questionário.

Questão 9: Módulos devem depender de abstrações e não de classes concretas e que módulos de alto nível não devem depender de módulos de baixo nível e que ambos devem depender de abstrações.
Programador 1: “Princípio da substituição de Liskov“
Programador 2: “Princípio da inversão de dependência”
Programador 3: “Baixo acoplamento”
Programador 4: “Princípio da inversão de dependência”
Programador 5: “Princípio da inversão de dependência”
Programador 6: “Princípio da inversão de dependência”
Programador 7: “Princípio da inversão de dependência”

A resposta correta para a questão 9 é o princípio da inversão de dependência. Apenas os programadores 1 e 3 erraram esta questão.

Tabela 11. Respostas da questão 10 do questionário.

Questão 10: Clientes não devem ser forçados a implementar interfaces que não usam ou

não devem ser forçados a depender de métodos que não usam.
Programador 1: “Princípio da segregação de interface”
Programador 2: “Princípio da segregação de interface”
Programador 3: “Princípio da segregação de interface”
Programador 4: “Princípio da segregação de interface”
Programador 5: “Princípio da segregação de interface”
Programador 6: “Princípio da segregação de interface”
Programador 7: “Princípio da segregação de interface”

Na questão 10 todos responderam corretamente.

Tabela 12. Respostas da questão 11 do questionário.

Questão 11: Quando uma classe está pouco conectada, possui pouco conhecimento ou depende de poucas classes.
Programador 1: “Baixo acoplamento”
Programador 2: “Baixo acoplamento”
Programador 3: “Princípio da inversão de dependência”
Programador 4: “Baixo acoplamento”
Programador 5: “Baixo acoplamento”
Programador 6: “Baixo acoplamento”
Programador 7: “Baixo acoplamento”

A resposta correta para a questão 11 é o baixo acoplamento. Apenas o programador 3 errou esta questão.

Tabela 13. Respostas da questão 12 do questionário.

Questão 12: É um objeto responsável por tratar eventos ou definir métodos para as operações do sistema. Serve para minimizar a dependência entre a interface de usuário e as classes de modelo de domínio.
Programador 1: “Controlador”
Programador 2: “Fabricação pura”
Programador 3: “Fabricação pura”
Programador 4: “Indireção”
Programador 5: “Fabricação pura”
Programador 6: “Indireção”
Programador 7: “Polimorfismo”

Na questão 12 houve uma confusão generalizada e de certa forma esperada. Esta confusão foi esperada pois é fácil confundir os princípios controlador, fabricação pura e indireção para quem está começando a aprender os princípios GRASP já que todos estes falam em desacoplamento (minimizar a dependência) e na criação de objetos intermediadores. E as respostas desta questão acabaram interferindo nas respostas dos princípios relacionados.

Outra observação importante, e que contribuiu muito para a confusão, é que este princípio não estava nos slides da apresentação e nada foi dito sobre ele. Este erro foi

percebido após a análise destas respostas e ele foi informado posteriormente aos participantes. Apenas o programador 1 acertou a resposta, que é o controlador.

Tabela 14. Respostas da questão 13 do questionário.

Questão 13: Permite que classes derivadas possam ser referenciadas por suas classes base e que métodos de uma classe base possam invocar métodos que possuem a mesma assinatura, porém com comportamentos distintos, em suas classes derivadas.
Programador 1: “Polimorfismo”
Programador 2: “Polimorfismo”
Programador 3: “Polimorfismo”
Programador 4: “Polimorfismo”
Programador 5: “Polimorfismo”
Programador 6: “Polimorfismo”
Programador 7: “Polimorfismo”

Na questão 13 todos responderam corretamente.

Tabela 15. Respostas da questão 14 do questionário.

Questão 14: Identificar os pontos de software que variam ou que possam sofrer uma instabilidade prevista e então separar estas variantes das partes que permanecem inalteradas em abstrações, de modo que agora o sistema dependa de uma abstração e não mais de algo concreto.
Programador 1: “Variações protegidas”
Programador 2: “Variações protegidas”
Programador 3: “Variações protegidas”
Programador 4: “Variações protegidas”
Programador 5: “Variações protegidas”
Programador 6: “Variações protegidas”
Programador 7: “Indireção”

Na questão 14 somente o programador 7 errou a resposta.

Tabela 16. Respostas da questão 15 do questionário.

Questão 15: Similar ao princípio da fabricação pura. Diminui o acoplamento direto entre objetos ao criar um objeto intermediário, maximizando o potencial de reuso. Um exemplo é o padrão MVC.
Programador 1: “Indireção”
Programador 2: “Controlador”
Programador 3: “Indireção”
Programador 4: “Especialista da informação”
Programador 5: “Controlador”
Programador 6: “Controlador”
Programador 7: “Controlador”

Na questão 15 ocorreu um problema similar à questão 12 do controlador: uma confusão esperada devido ao fato de ser fácil confundir os princípios controlador,

fabricação pura e indireção para quem está começando a estudar os princípios GRASP. Somente os programadores 1 e 3 acertaram a resposta correta que é indireção.

Tabela 17. Respostas da questão 16 do questionário.

Questão 16: Consiste na criação de novas classes para diminuir o acoplamento e aumentar a coesão de classes.
Programador 1: “Especialista da informação”
Programador 2: “Criador”
Programador 3: “Criador”
Programador 4: “Criador”
Programador 5: “Especialista da informação”
Programador 6: “Fabricação pura”
Programador 7: “Criador”

A questão 16 é similar à situação das questões 12 e 15. Somente o programador 6 respondeu corretamente.

Tabela 18. Respostas da questão 17 do questionário.

Questão 17: Ajuda na decisão sobre qual classe deve ser responsável pela criação de novas instâncias de objetos.
Programador 1: “Criador”
Programador 2: “Indireção”
Programador 3: “Controlador”
Programador 4: “Controlador”
Programador 5: “Criador”
Programador 6: “Criador”
Programador 7: “Fabricação pura”

A resposta correta para a questão 17 é o padrão criador, ou seja, nesta questão estão corretos os programadores 1, 5 e 6.

Tabela 19. Respostas da questão 18 do questionário.

Questão 18: Oferece instruções sobre qual classe se deve atribuir uma responsabilidade. Basicamente, ele diz que uma responsabilidade A deve ser atribuída à classe B se B possui as informações necessárias para que a responsabilidade A seja executada.
Programador 1: “Fabricação pura”
Programador 2: “Especialista da informação”
Programador 3: “Especialista da informação”
Programador 4: “Fabricação pura”
Programador 5: “Indireção”
Programador 6: “Especialista da informação”
Programador 7: “Especialista da informação”

Na questão 18, última questão do questionário, a resposta correta é especialista da informação, ou seja, estão corretos os programadores 2, 3, 6 e 7.

Sumarizando os erros e acertos das questões objetivas encontra-se o seguinte cenário:

Tabela 20. Sumário de erros e acertos das questões objetivas.

Programador	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18
1	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✗	✓	✗
2	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗	✓
3	✓	✓	✓	✗	✓	✗	✗	✓	✓	✓	✗	✗	✓
4	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗	✗
5	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	✓	✗
6	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓
7	✓	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓

De um total de 91 respostas, 62 foram corretas e 29 incorretas, um seja uma porcentagem de acerto de 68,13%, que pode ser considerada uma boa porcentagem tendo em vista que foi o primeiro contato da maioria dos programadores quanto aos princípios apresentados e houve uma confusão em uma das questões, o que acabou afetando um pouco as demais. Mas como dito anteriormente, o objetivo das questões objetivas foi fazer com que os programadores parassem para refletir sobre os ensinamentos e tentassem melhor absorvê-los. Com o uso do *checklist* e com a cultura de boas práticas cada vez mais presente na equipe, a tendência é que estes princípios se tornem cada mais familiares aos programadores até que eles os dominem.

6. Considerações finais

Esta pesquisa teve como objetivo principal aprimorar o conhecimento coletivo dos sujeitos participantes, no caso a equipe de programadores da unidade de análise, quanto a princípios de codificação voltados a manutenção de sistemas, para que então estes possam escrever códigos melhores em termos de manutenção. Para alcançar isto, foram definidos alguns objetivos específicos:

- Com a pesquisa bibliográfica, avaliar princípios de design referentes a linguagens de programação orientadas a objetos voltados a manutenção de sistemas de software, como SOLID e GRASP.
- Refatorar códigos de produção com a aplicação destes princípios, para complementar o estudo e servir de exemplo aos programadores.
- Elaborar um *checklist* de boas práticas de codificação para servir de auxílio aos programadores durante o desenvolvimento e para absorção gradual dos princípios estudados.
- Elaborar e realizar um *workshop* com o intuito de passar este conhecimento adquirido aos programadores da equipe alvo e apresentar o *checklist* de boas práticas.
- Elaborar e realizar de um questionário para coleta de *feedback* dos programadores.

Analisando os dados qualitativos e os resultados da pesquisa, pode-se dizer que ela alcançou todos os objetivos estabelecidos com sucesso. Com a pesquisa bibliográfica, foi possível avaliar e aprimorar o conhecimento quanto a princípios

voltados a manutenção de sistemas, princípios estes que visam criar um sistema mais fácil de ser mantido e menos frágil a mudanças. Com o estudo sobre estes princípios concluído, foi possível analisar e refatorar códigos em produção do sistema Web da empresa alvo, que serviram para complementar os exemplos do estudo e também para que os programadores tivessem exemplos familiares e pudessem compreender melhor os princípios apresentados. Percebe-se pelo questionário que isto contribuiu muito para o entendimento. Com a elaboração e apresentação do *workshop*, foi possível apresentar a equipe todos os princípios estudados, com exemplos simples e didáticos, e para complementar, também foi possível apresentar os códigos refatorados do sistema em produção e o *checklist* de boas práticas criado para criar uma cultura de boas práticas dentro da equipe.

Este trabalho pode contribuir muito a todos que desejam aprimorar os seus conhecimentos, e das suas equipes de desenvolvimento, quanto a princípios de programação orientada a objetos voltados a manutenção de sistemas. Nele estão inclusos, além da definição destes princípios, exemplos didáticos e de fácil compreensão, que auxiliam na sua compreensão. Um dos membros da equipe comentou em uma conversa posterior que já havia ouvido falar de alguns dos princípios, mas que os exemplos que ele encontrara não fizeram com que ele entendesse plenamente suas aplicações, e com os exemplos deste trabalho ele pôde. Estes exemplos (Apêndice D), de autoria do autor deste trabalho, estão disponíveis publicamente em um repositório do *GitHub* e podem ser acessados por todos que desejam compreender melhor os princípios, com o antes e depois de suas aplicações. Além dos códigos no *GitHub*, estão disponíveis publicamente no *OneDrive* o *checklist* de boas práticas (Apêndice A) e os *slides* do *workshop* (Apêndice B).

Este trabalho também demonstrou que o *workshop* é uma ótima forma de transmitir conhecimento às equipes de desenvolvimento. Isto pôde ser percebido nas respostas do questionário e no *feedback* da equipe em conversas posteriores. Todos os membros da equipe, inclusive o próprio proprietário e diretor executivo da empresa, elogiaram a forma como o *workshop* foi apresentado, com definições claras e objetivas, com uso de exemplos simples e de códigos refatorados, e já estão organizando futuros *workshops* para que seja criado uma oportunidade aos membros da equipe de trazer novos conteúdos e aprimorar o conhecimento coletivo.

Além disto, este trabalho demonstrou que a criação e utilização de um *checklist* de boas práticas pode ser uma ótima forma de tornar a utilização destes princípios algo cultural dentro de uma equipe. Nos dias que se seguiram após a apresentação, vários membros da equipe começaram a refletir sobre os códigos que estavam produzindo e que iriam produzir, recorrendo a estes princípios.

Algumas limitações deste trabalho encontraram-se no fato da empresa e da equipe não possuírem até o momento da apresentação a cultura de criar e organizar oportunidades para que os membros da equipe possam estudar e trazer novos conhecimentos. As empresas como um todo precisam compreender que estas oportunidades geram inúmeros benefícios às equipes de trabalho, não somente equipes de desenvolvimento. Estes momentos são de vital importância para que as equipes possam compartilhar conhecimento e trazer novidades que podem ser utilizadas em projetos futuros. Uma das dificuldades encontradas neste trabalho foi a disponibilidade de tempo dentro do horário de expediente para a realização do *workshop*. Mas ao final

da apresentação, após o retorno positivo por parte da equipe, esta percebeu a importância de se realizar e organizar estes eventos e todos concordaram em tornar isto uma prática.

Para trabalhos futuros, pode-se abranger mais sobre boas práticas de desenvolvimento além dos princípios voltados a manutenção de sistemas apresentados neste trabalho. Um dos itens do questionário referiu-se justamente a estes trabalhos futuros, para que os membros da equipe começassem a pensar no que poderia ser apresentado em futuros *workshops*, e as respostas foram ótimas. Pode-se por exemplo dar continuidade a este trabalho abrangendo-se sobre os *design patterns* apresentados por Gamma e outros (GAMMA, HELM, JOHNSON, VLISSIDES, 1994), sobre código limpo, e outros princípios como KISS, YAGNI, DRY, *package*, etc. Além disto, pode-se abranger sobre os diferentes tipos de manutenção que podem ocorrer em um *software*, como manutenção corretiva, adaptativa, evolutiva e preventiva, e onde cada princípio abordado pode agir sobre elas.

A chave fundamental de um código bem escrito, o santo grau do desenvolvimento: permitir que se possa ampliar, corrigir ou alterar o código existente de um sistema sem impactar seu comportamento, criando ou alterando somente o código necessário e em apenas um local. Esta não é uma tarefa simples, porém, é facilitada quando o programador tem conhecimento sobre princípios que auxiliam nesta tarefa. Com o conhecimento adquirido com esta pesquisa, a equipe alvo de programadores passou a conhecer mais sobre estes princípios e foi dado o primeiro passo para a criação de uma cultura organizacional pensada em boas práticas.

Referências

GAMMA, Erich. (2000), Padrões de Projeto: soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman 1. ed.

GIL, Antonio Carlos. (2008), Métodos e técnicas de pesquisa social. São Paulo: Atlas, 6. ed.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. (1994), Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1. Ed.

LAKATOS, Eva Maria; MARCONI, Marina de Andrade. (2009), Técnicas de pesquisa: planejamento e execução de pesquisas, amostragens e técnicas de pesquisas, elaboração, análise e interpretação de dados. São Paulo: Atlas, 7. ed.

LARMAN, Craig. (2005), Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Pearson Education International, 3. ed.

MARTIN, Robert C. (2000), Design Principles and Design Patterns. Disponível em: http://staff.cs.utu.fi/staff/jouni.smed/doos_06/material/DesignPrinciplesAndPatterns.pdf. Acesso em: 21 ago. 2019.

MARTIN, Robert C. (2002), Agile Software Development, Principles, Patterns, and Practices. Pearson, 1. ed.

ROESCH, Sylvia Maria Azevedo. (2009), Projetos de estágio e de pesquisa em administração: guia para estágios, trabalhos de conclusão, dissertações e estudos de caso. São Paulo: Atlas, 3. ed.

SILVA, Lucas Rafael da. (2016), Necessidade de se utilizar boas práticas arquiteturais e padrões de projeto no desenvolvimento de web service baseado na arquitetura restful. Faculdades Integradas de Caratinga, Minas Gerais.

SOMMERVILLE, Ian. (2011), Engenharia de Software. São Paulo: Pearson, 9. ed.

SOUZA, Virginia Farias. (2018), Uma Análise do Impacto de Code Smells nas Mudanças de Software. Pernambuco: Centro de Estudos e Sistemas Avançados do Recife – CESAR.

TORRES, José Jorge Barreto. (2016), Identificação e análise de clones de códigos heterogêneos em um ambiente corporativo de desenvolvimento de software. Sergipe: Universidade Federal de Sergipe (UFS).

YIN, Robert K. (2010), Estudo de caso: planejamento e métodos. Porto Alegre: Bookman, 4. ed.

ANEXO A – MÉTODO GENERATECONTEXT DA CLASSE PRODUCTSERVICE

```
public void GenerateContext(BudgetProduct product)
    {
        Product productBase;
        List<Field> fields;
        List<Translation> translations;
        List<Process> processes;
        List<MeasurementUnit> units;
        List<Feedstock> feedstocks;
        List<string> localVariables;
        List<Checklist> checklists;

        LoadContextInformations(product, out productBase, out fields, out
translations, out processes, out units, out feedstocks, out checklists);

        try
        {
            GenerateProductContext(product, productBase, fields, translations,
units, out localVariables, checklists);
        }
        catch (WarningException exception)
        {
            throw new WarningException(exception.MessageTranlationKey);
        }

        foreach (BudgetProcess process in product.Processes)
        {
            Process processBase = processes.Find(p => p.PublicId ==
process.PublicId && !p.Lock);
            try
            {
                ProcessService.GenerateContext(process, processBase, fields,
translations, units, localVariables);
            }
            catch (WarningException exception)
            {
                throw new WarningException(exception.MessageTranlationKey);
            }
        }

        foreach(FeedstockFilter feedstockFilter in product.Feedstocks)
        {
            try
            {
                FeedstockService.GenerateContext(feedstockFilter, units);
            }
            catch (WarningException exception)
            {
                throw new WarningException(exception.MessageTranlationKey);
            }
        }
    }
```

ANEXO B – MÉTODO GETBYTYPE DA CLASSE FIELDSERVICE

```
private FieldContent GetByType(FieldContentType fieldContentType)
{
    switch (fieldContentType)
    {
        case FieldContentType.checkboxOptions:
            return new HoldprintArrayContent(new List<string>());
        case FieldContentType.time:
            return new HoldprintTimeContent(string.Empty);
        case FieldContentType.intervalTime:
            return new HoldprintIntervalTimeContent(string.Empty);
        case FieldContentType.date:
            return new HoldprintDateContent();
        case FieldContentType.intervalDate:
            return new HoldprintDateIntervalContent();
        case FieldContentType.texto:
            return new HoldprintTextContent(string.Empty);
        case FieldContentType.longtext:
            return new HoldprintTextContent(string.Empty);
        case FieldContentType.autocomplete:
            return new HoldprintArrayContent(new List<string>());
        case FieldContentType.radioOptions:
            return new HoldprintTextContent(string.Empty);
        case FieldContentType.phone:
            return new HoldprintTextContent(string.Empty);
        case FieldContentType.feedstockFilter:
            return new HoldprintFeedstockFilterContent();
        case FieldContentType.inkTypeSelector:
            return new HoldprintNumberContent(0);
        case FieldContentType.switchButton:
            return new HoldprintBooleanContent(false);
        default:
            return new HoldprintTextContent(string.Empty);
    }
}
```

ANEXO C – INTERFACE IPROCESSSERVICE

```
public interface IProcessService : IService
{
    Process Create(Process process);

    Process Find(string id);

    Process Find(int publicId);

    Process Find(int publicId, int version);

    List<Process> FindByMecId(int mecPublicId);

    List<Process> FindProcessesFromPublicIds(string params_);

    List<Process> FindProcessesFromPublicIds(List<int> publicIds);

    List<Process> FindProcessesFromList(string[] ids);

    IEnumerable<Process> FindByFilters(List<IHoldprintFilter> filters, int offset, int
page);
```

```
    Process Update(Process process);

    Process AddChecklist(Checklist checklist, string id);

    Process AddWarning(Warning warning, string id);

    Process AddCustomField(Field field, string id);

    Process Delete(int publicId);

    RelatedProcess UpdateByVariableModel(ProcessVariableModel pvm, RelatedProcess
process);

    List<Process> List();

    Process DiscoveryChecklist(Process process);

    Process Patch(JsonPatchDocument<Process> patchItem, int publicId, string id);

    List<ProcessCostEngineeringModel> ListProcessCostEngineeringModels();

    ProcessCostEngineeringModel GetProcessCostEngineeringModel(string id);

    List<ProcessCostEngineeringModel> ReadProcessCostEngineeringModelsWithStatus();

    Task<object> ProcessCostEngineeringModelGraphQL(GraphQLQuery query);

    Task<object> ProcessGraphQL(GraphQLQuery query);

    ProcessCostEngineeringModel UpdateProcessCostEngineeringModelVersion(int publicId);

    ProcessCostEngineeringModel ImportProcessCostEngineeringModel(int publicId);

    List<ProcessCostEngineeringModel> UpdateAllProcessCostEngineeringModels();

    List<ProcessFamily> ListFamilies();

    List<ProcessFamily> ListParentFamilies();

    ProcessFamily CreateFamily(ProcessFamily processFamily);

    BudgetProcess GetBudgetProcess(RelatedProcess relatedProcess);

    List<Process> LoadProcessFromRelatedProcess(List<RelatedProcess> relatedProcesses);

    BudgetProcess GetBudgetProcess(Process Process);

    BudgetProcess LoadScripts(BudgetProcess process);

    void GenerateContext(BudgetProcess budgetProcess, Process process, List<Field>
fields, List<Translation> translation, List<MeasurementUnit> units, List<string>
receiveVariables);

    UserPreference ChangeBookmark(int publicId);

    ProcessCostEngineeringModel GetProcessCostEngineeringModelByPublicId(int publicId);

    Process ResolveChecklist(Process process);
  }
```

ANEXO D – CLASSE CITIESFACTORY

```
public class CitiesFactory
{
    private readonly ApplicationContext applicationContext;

    public CitiesFactory(ApplicationContext applicationContext)
    {
        this.applicationContext = applicationContext;
    }

    public async Task Create(City city)
    {
        applicationContext.Cities.Add(city);
        await applicationContext.SaveChangesAsync();
    }

    public async Task<List<City>> Read()
    {
        return await applicationContext.Cities.ToListAsync();
    }

    public async Task<City> Read(int id)
    {
        return await applicationContext.Cities.FindAsync(id);
    }

    public async Task<City> GetCityByIbgeId(int idIbge)
    {
        return await applicationContext.Cities.FirstOrDefaultAsync(x =>
x.SpecificCountryCode == idIbge);
    }
}
```

ANEXO E – CONTROLADOR CITIESCONTROLLER

```
public class CitiesController : ApiController
{
    private readonly ApplicationContext applicationContext = new ApplicationContext();

    public async Task<List<City>> GetCities()
    {
        var citiesFactory = new CitiesFactory(applicationContext);
        return await citiesFactory.Read();
    }
}
```

ANEXO F – CLASSE BUDGETCALCCONSUMER

```
public class BudgetCalcConsumer
{

    private static readonly DefaultContractResolver contractResolver = new
DefaultContractResolver
        {
            NamingStrategy = new SnakeCaseNamingStrategy()
        };

    public static void ConsumeCalcStatus(BasicDeliverEventArgs ea)
    {
        string message = Encoding.UTF8.GetString(ea.Body);
        CalcSnippet calcSnippet = JsonConvert.DeserializeObject<CalcSnippet>(message, new
JsonSerializerSettings
{
    ContractResolver = contractResolver,
    Formatting = Formatting.Indented
});

        using (var scope =
((AutofacDependencyResolver)Startup.DependencyResolver).Container.BeginLifetimeScope(buil
der =>
        {

builder.RegisterType(typeof(AccountMongoApplicationContext)).As(typeof(IAccountMongoAppli
cationContext))
                .InstancePerLifetimeScope()
                .WithParameter(
                    new ResolvedParameter(
                        (pi, ctx) => pi.ParameterType == typeof(Func<int>),
                        (pi, ctx) => new Func<int>(() => calcSnippet.AccountId)))
                .WithParameter(
                    new ResolvedParameter(
                        (pi, ctx) => pi.ParameterType == typeof(Func<string>),
                        (pi, ctx) => new Func<string>(() => string.Empty)));
        }))
        {
            scope.Resolve<ICalculationService>().UpdateCalc(calcSnippet);
        }
    }
}
```

APÊNDICE A – CHECKLIST DE BOAS PRÁTICAS

https://1drv.ms/w/s!Ah5YCrJt1dJsi6AT1M7nXUPzbOAglQ?e=OGXL7G

APÊNDICE B – SLIDES DO WORKSHOP

https://1drv.ms/p/s!Ah5YCrJt1dJsi6ARV8pWlAzQSJ1mSA?e=u0a9Sb

APÊNDICE C – QUESTIONÁRIO

https://1drv.ms/w/s!Ah5YCrJt1dJsi6AgNWUFkoVonSp0Rg?e=uApH3i

APÊNDICE D – CÓDIGOS DE EXEMPLO

https://github.com/Formagio/BonsPrincipiosPraticas