UNICESUMAR - CENTRO UNIVERSITÁRIO CESUMAR
PROGRAMA DE PÓS-GRADUAÇÃO DE PROGRAMAÇÃO ORIENTADA A OBJETOS EM JAVA

UMA CONTRIBUIÇÃO EMPÍRICA PARA RENASCER PRODUTOS DE SOFTWARE

JULIANO RIBEIRO DE SOUZA MACIEL

MARINGÁ
2020

JULIANO RIBEIRO DE SOUZA MACIEL

TRABALHO DE CONCLUSÃO DE CURSO

UMA CONTRIBUIÇÃO EMPÍRICA PARA RENASCER PRODUTOS DE SOFTWARE

Trabalho de conclusão de curso apresentado ao Centro Universitário Cesumar (UNICESUMAR), como requisito à obtenção do título de Especialista em Programação Orientada a Objetos em JAVA.
Linha de Pesquisa: Criação e Manutenção em Desenvolvimento de Software.
Orientador: *Msc. Arthur Cattaneo Zavadski*

MARINGÁ

OUTUBRO DE 2020

AGRADECIMENTOS

Primeiramente aos meu pais Sr. Juarez e Sra. Elizabete e a minha irmã Monique que me incentivaram a chegar até o presente momento com muita persistência e resiliência. Ao meu amigo Jeremias com o qual dividi inúmeros momentos durante essa jornada de quase três anos, com chuva, sol ou frio, mas enfim estamos encerrando este ciclo. A minha prima Jeane, que além de me incentivar, auxiliou na formatação do presente trabalho. Enfim a todos os mestres e professores que se passaram até hoje na minha vida acadêmica.

SUMÁRIO

1. Resumo

O presente trabalho teve como objetivo, estudar, de maneira breve, *frameworks* para desenvolvimento de produtos de *software*, consolidados ao longo das décadas, a fim de auxiliar desenvolvedores de *software* na construção ou refatoração de soluções de *software* com objetivo de as tornarem manuteníveis ao médio / longo prazo.

Palavras-chave: Produto; Software; Padrões de Projeto; Boas Práticas de Programação; Refatoração; Testes Unitários Automatizados.

2. Introdução

O desenvolvimento de *software* é uma das áreas mais desafiadoras onde um profissional pode atuar, baseio tal afirmação em uma palavra utilizada nesta área, a heurística. Em suma, dois profissionais de mesmo nível, onde são requisitados para fazer a mesma tarefa, com as mesmas ferramentas podem efetuar tal procedimento com construções totalmente diferentes, mas que proporcionam o mesmo resultado.

Nesta perspectiva, diante da probabilidade exponencial de construções para se obter um mesmo resultado em uma implementação de produto de *software*, percebe-se a necessidade de avaliar a aplicação de padrões de projeto em produtos de *software* não idealizados com tal, a fim de reverter o cenário de um *software* espólio em produto rentável a médio / longo prazo.

Além disso, outros *frameworks*, podem contribuir na manutenibilidade do produto, como a aplicação de boas práticas de programação, testes unitários automatizados e não menos importante a refatoração, pois um *software* assim como nós está em constante evolução.

3. Desenvolvimento

3.1. Padrões de Projeto

Um padrão de projeto, visa solucionar um problema comum entre produtos de *software*, abstraindo sua complexidade e propondo uma solução que possa ser aplicada em contextos diferentes de aplicações, contudo não garantindo a qualidade do código fonte resultante.

Segundo Cornélio (2004), os padrões de projeto registram conhecimento e experiência adquiridos ao longo das décadas de desenvolvimento de *software*, sendo criados a partir de observações e experimentação, o resultado deste processo pode ser replicado durante a modelagem ou codificação no decorrer do processo de criação de um produto de *software*.

Já Coplien (1996) diz que cada padrão de projeto é uma regra de três partes, sendo a primeira uma expressão de uma relação de um determinado contexto, a segunda um problema e a terceira a própria solução. Um padrão de projeto, é uma coisa, que acontece no mundo ao mesmo tempo uma regra que nos diz como e quando criá lo, ou seja, descreve algo que está vivo, e ao mesmo tempo descreve seu processo de criação.

Para identificar ou criar um padrão de projeto, seguimos algumas características básicas, sendo elas, *Nome*, para identificação fácil e simples, *Problema*, o que ele resolve, *Solução*, como ele resolve e *Consequências*, quais são os contras ao aplicá-lo. Logo sendo eles a experiência e lições aprendidas por outros desenvolvedores, como podemos absorver tal conhecimento e colocá lo em prática? Freeman et al. (2014) propõe que a melhor maneira é carregar seu cérebro com eles e, em seguida, reconhecer lugares seja em novas ou já existentes soluções que possam ser aplicados, em vez de reutilização de código, você obtém a reutilização de experiência.

> Os padrões de projeto podem também variar em sua granularidade e nível de abstração, resultando em uma grande diversificação, sendo assim é preciso organizá-los. A classificação ajuda a compreender os padrões de projeto de forma eficaz, além de direcionar de forma simples os esforços necessários para o desenvolvimento de novos padrões de projeto (GAMMA et al., 1994).

O conceito de catálogo de para classificação de padrões de projeto foi apresentado pela literatura formal, em 1977 por Christopher Alexander, entretanto, o

mesmo não pertencia a indústria de *software*, sendo ele arquiteto, fez uma compilação com cerca de duzentos e cinquenta padrões voltados para a construção civil. A partir do trabalho de Alexander a primeira proposta prática apresentada para o desenvolvimento de *software* foi a publicação Gamma et al. (1994), sendo a principal referência no assunto para a comunidade de *software*, descrevendo vinte e três padrões baseados nas experiências durante a carreira de desenvolvedor de *software*.

Importante destacar que daqui a diante o texto se refere a padrões de projeto no paradigma orientado a objetos[1]. O primeiro critério para a classificação de um padrão de projeto, é a sua finalidade, de acordo com aspectos comuns entre padrões temos três categorias:

- Criacional: Para criarmos um objeto durante a execução do *software* é necessário instanciá-lo, um padrão de projeto do tipo criacional encapsula a criação do objeto, desta forma o mesmo deixa de ser instanciado diretamente pelo desenvolvedor que passa a acionar a classe responsável pelo mesmo.
- Estrutural: Define a composição de uma classe ou objeto, a partir de herança ou implementação de interfaces, a fim de obter, comportamentos distintos ou polimorfismo, para rotinas e funcionalidades do *software*.
- Comportamental: São especializados em comunicação entre objetos.

Dentro destas três categorias temos padrões de classe, voltados para a herança entre superclasses e subclasses, portanto são estáticos, e padrões de objeto que são especialistas em relacionamentos que sofrem polimorfismo, logo são dinamicamente alterados em tempo de execução do *software*.

Para dividir os padrões de projeto devidamente em suas categorias, o segundo critério, se remete a seu escopo, especificamente a aplicabilidade do padrão a uma classe ou objeto, sendo distribuídos da seguinte forma:

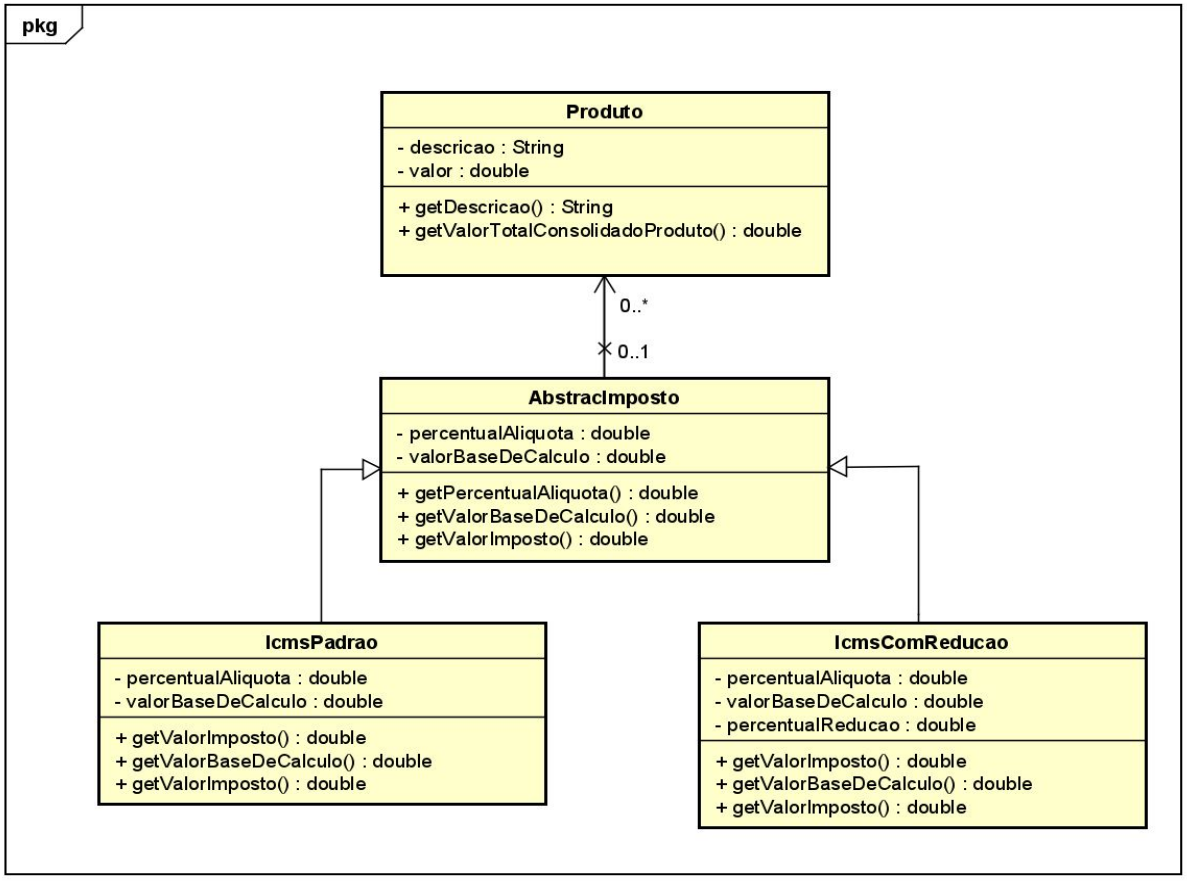
- Criacionais: *Factory Method, Abstract Factory, Builder, Prototype* e *singleton.*
- Estruturais: *Adapter, Bridge, Composite, Decorator, Facade, Flyweight* e *Proxy*.

[1] Daqui em diante serão apresentados conforme seus aspectos e classificações definidos por Gamma et al. (1994). Para uma visão diferente da aqui apresentada sugiro a publicação de Coplien (1996).

- Comportamentais: *Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method* e *Visitor*.

O exemplo a seguir implementa o padrão de projeto *Template Method*, sendo do tipo comportamental, onde é definido no núcleo da superclasse métodos estratégicos como abstratos a fim de delegar as subclasses a responsabilidade de implementar de suas particularidades.

Figura 1: Diagrama de classe *template method*.



Fonte: MACIEL, Juliano Ribeiro de Souza 2020.

Figura 2: Implementação da classe *Produto*

```java
package org.tcc.classes;

import java.util.List;

public class Produto {
    private String descricao;
    private double valor;
    private List<AbstractImposto> impostos;

    public Produto(String descricao, double valor, List<AbstractImposto> impostos) {
        super();
        this.descricao = descricao;
        this.valor = valor;
        this.impostos = impostos;
    }

    private double getValorTotalImpostos(){
        return impostos.stream().mapToDouble(x -> x.getValorImposto()).sum();
    }

    public String getDescricao() {
        return this.descricao;
    }

    public double getValorTotalConsolidadoProduto() {
        return this.valor - getValorTotalImpostos();
    }
}
```

Fonte: MACIEL, Juliano Ribeiro de Souza 2020.

Figura 3: Implementação da classe *AbstracImposto*

```java
package org.tcc.classes;

public abstract class AbstractImposto {
    private double prAliq;
    private double vlBaseCalc;

    public AbstractImposto(double percentualAliquota, double valorBaseDeCalculo) {
        this.prAliq = percentualAliquota;
        this.vlBaseCalc = valorBaseDeCalculo;
    }

    protected double getPercentualAliquota() {
        return this.prAliq;
    }

    protected double getValorBaseDeCalculo() {
        return this.vlBaseCalc;
    }

    public abstract double getValorImposto();
}
```

Fonte: MACIEL, Juliano Ribeiro de Souza 2020.

Figura 4: Classe concreta *IcmsPadrao* que implementa a classe *AbstracImposto*

```java
package org.tcc.classes;

public class IcmsPadrao extends AbstractImposto{

    public IcmsPadrao(double percentualAliquota, double valorBaseDeCalculo) {
        super(percentualAliquota, valorBaseDeCalculo);

        if(this.getPercentualAliquota() <= 0)
            throw new IllegalArgumentException("Percentual Alíquota inválido! Verifique.");
        else if(this.getValorBaseDeCalculo() <= 0)
            throw new IllegalArgumentException("Valor Base de Cálculo inválido! Verifique.");
    }

    @Override
    public double getValorImposto() {
        return getValorBaseDeCalculo() * (this.getPercentualAliquota() / 100);
    }
}
```

Fonte: MACIEL, Juliano Ribeiro de Souza 2020.

Figura 5: Classe concreta *IcmsComReducao* que implementa a classe *AbstractImposto*

```java
package org.tcc.classes;

public class IcmsComReducao extends AbstractImposto {

    private double percentualDeReducao;

    public IcmsComReducao(double percentualAliquota, double percentualReducao, double valorBaseCalculo) {
        super(percentualAliquota, valorBaseCalculo);
        this.percentualDeReducao = percentualReducao;

        if(this.getPercentualAliquota() <= 0)
            throw new IllegalArgumentException("Percentual Alíquota inválido! Verifique.");
        else if(this.getValorBaseDeCalculo() <= 0)
            throw new IllegalArgumentException("Valor Base de Cálculo inválido! Verifique.");
        else if(this.percentualDeReducao <= 0)
            throw new IllegalArgumentException("Percentual de redução inválido! Verifique.");
    }

    @Override
    public double getValorImposto() {
        return getValorBaseDeCalculo()
                * (1 - ((getPercentualAliquota() / 100) * (1 - this.percentualDeReducao)))
                / (1 - (getPercentualAliquota() / 100)) - getValorBaseDeCalculo();
    }
}
```

Fonte: MACIEL, Juliano Ribeiro de Souza 2020.

No exemplo apresentado a classe *AbstractImposto* define que todas suas subclasses deverão implementar o método *getValorImposto*, este é núcleo do algoritmo para que na classe produto seja calculado o valor total consolidado, sendo ele, a diferença entre seu valor total líquido e o valor total dos impostos. A centralização de códigos comuns nas superclasses é uma das vantagens deste

padrão de projeto, porém quando um algoritmo possui muitas etapas delegadas as subclasses torna se mais complexa sua manutenção.

> A lição a se tirar, no que diz respeito a implementar padrões de projeto[2], é que os objetos podem ser idealizados e bem definidas suas responsabilidades, além de fornecerem um conjunto explicável de idiomas, por meio dos quais podem ser construídos sistemas bem projetados (LARMAR, 2007).

3.2. Boas Práticas de Programação

Conforme descrito na seção passada, a aplicação de padrões de projeto, lhe garante o funcionamento de determinada funcionalidade sendo ela provada por ter sido aplicada em uma série de aplicações com contextos distintos, logo é a solução de um problema para um determinado escopo, contudo não é correto afirmar que a forma como esta implementação foi realizada pelo desenvolvedor é de alta qualidade, ou seja, a legibilidade e manutenibilidade será facilmente aplicada por outros desenvolvedores ao longo da vida do produto de *software*, para isto ao longo das décadas foram apresentadas boas práticas para desenvolvedores construírem suas soluções.

> Código limpo é simples e direto. ele é tão bem legível quanto uma prosa bem escrita. Ele jamais torna confuso o objetivo do desenvolvedor em vez disso ele está repleto de abstrações claras e linhas de controle objetivas. Um código confuso tem seu custo, e quanto mais a confusão aumenta, menor a produtividade dos desenvolvedores para mantê-lo ou adicionar novas funcionalidades (MARTIN, 2008).

Uma classe é uma coleção de dados e rotinas que compartilham uma responsabilidade coesa e bem definida. Uma classe também pode ser uma coleção de rotinas que fornece um conjunto coeso de serviços, mesmo que nenhum dado comum esteja envolvido (McConnell, 2004). Ao iniciar um projeto de *software*, uma das primeiras coisas feitas é nomeá lo, essa é uma constância do programador seja criando classes, métodos ou variáveis, então fazer isso de forma consciente trás benefícios ao código no médio/longo prazo. Martin (2008) diz que, os nomes utilizados em uma aplicação devem revelar sua intenção, o que está sendo proposto naquele contexto, por que ele existe, o que faz e como é usado. Se um nome requer

[2] Mais exemplos de padrões de projeto, podem ser encontrados no site *Refactoring.* Disponível em: <https://refactoring.guru/pt-br/design-patterns/catalog>.

um comentário, este não revela sua intenção. Este pequeno *framework*[3] ajuda o código a ser legível e manutenível, dando longevidade ao produto de *software*.

Na (figura 6) notamos que a classe implementa o padrão de projeto *template method*, porém não segue uma boa prática em relação a nomenclatura de suas variáveis, na (figura 7), é apresentada a classe após aplicação do *framework*.

Figura 6: Superclasse *AbstracImposto* pré refatoração de boas práticas de nomenclatura em suas variáveis

```
1  package org.tcc.classes;
2
3  public abstract class AbstractImposto {
4      private double prAliq;
5      private double vlBaseCalc;
6
7      public AbstractImposto(double percentualAliquota, double valorBaseDeCalculo) {
8          this.prAliq = percentualAliquota;
9          this.vlBaseCalc = valorBaseDeCalculo;
10     }
11
12     protected double getPercentualAliquota() {
13         return this.prAliq;
14     }
15
16     protected double getValorBaseDeCalculo() {
17         return this.vlBaseCalc;
18     }
19
20     public abstract double getValorImposto();
21 }
```

Fonte: MACIEL, Juliano Ribeiro de Souza 2020.

[3] Daqui em diante serão apresentados um conjunto frameworks elaborados na publicação de Martin (2008).

Figura 7: Superclasse *AbstracImposto* pós refatoração de boas práticas de nomenclatura em suas variáveis

```java
package org.tcc.classes;

public abstract class AbstractImposto {
    private double percentualAliquota;
    private double valorBaseDeCalculo;

    public AbstractImposto(double percentualAliquota, double valorBaseDeCalculo) {
        this.percentualAliquota = percentualAliquota;
        this.valorBaseDeCalculo = valorBaseDeCalculo;
    }

    protected double getPercentualAliquota() {
        return this.percentualAliquota;
    }

    protected double getValorBaseDeCalculo() {
        return this.valorBaseDeCalculo;
    }

    public abstract double getValorImposto();
}
```

Fonte: MACIEL, Juliano Ribeiro de Souza 2020.

Além da nomenclatura, serão apresentados a seguir três *framework's* de boas práticas de formatação de código:

- Formatação Vertical, onde as dependências de chamada de função apontem na direção descendente. Ou seja, uma função que é chamada deve estar abaixo de uma função que faz a chamada. Isso cria um bom fluxo descendo o módulo do código-fonte de alto a baixo nível.
- Formatação Horizontal, não é determinado um valor fixo, porém ao examinar sete projetos distintos, uma média de caracteres por linha fica entre 20 a 60, é proposto o valor de 80 como margem ideal em projetos de *software*.
- Indentação, são recuos incluídos em arquivos de código fonte, há informações que pertencem ao arquivo como um todo, às classes individuais dentro do arquivo, aos métodos dentro das classes, aos blocos dentro dos métodos e recursivamente aos blocos dentro dos blocos, recuamos as linhas do código-fonte em proporção à sua posição para tornar visível essa hierarquia.

Outra forma de manter o código saudável são as convenções de código. Elas são importantes pois oitenta por cento do custo de um *software* ao longo de seu

ciclo de vida é destinado para sua manutenção, e quase nenhum *software* é mantido por toda a sua vida pelo autor original Neste contexto as convenções de código melhoram a legibilidade do *software*, permitindo que os desenvolvedores compreendam o código a ser mantido de maneira mais rápida e completa. Ao oferecer o código-fonte de um *software* como um produto, há a necessidade de que ele esteja tão bem embalado e limpo quanto qualquer outro produto que possa ser avaliado fisicamente (ORACLE, 1999). Uma convenção define nome de arquivos, sufixos de arquivos, organização de arquivos, indentação, declarações de classes e interfaces, dentre outra séries de boas práticas para criação de um produto de *software* de alta qualidade.

3.3. Testes Unitários Automatizados

Até aqui foi apresentado como utilizar do conhecimento agregado e validado por padrões de projeto e como implementá los os com alto padrão de qualidade de código, utilizando boas práticas de programação, o próximo nível de qualidade do produto de *software* é garantir que qualquer desenvolvedor possa iniciar no projeto e implementar novas funcionalidades ou corrigir *bug's*, tendo um respaldo de que não afetará outros prontos da solução. Este respaldo podemos obter por meio de testes de unidade.

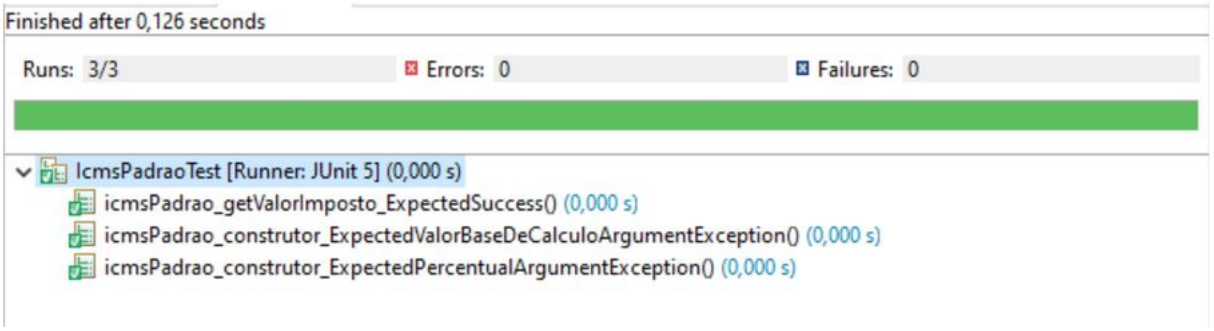
Um teste de unidade é um teste automatizado que, valida um pequeno trecho de código, de forma eficaz e isolada (KHORIKOV, 2020), ele deve ser auto suficiente (FOWLER et. al., 2014), invocando uma parte do *software* e validando seu resultado, se as suposições do resultado final estão erradas, o teste de unidade falhou, além disso deve ser confiável, legível e sustentável, podendo abranger tão pouco quanto um método ou tanto quanto várias classes (OSHEROVE, 2014). Um conjunto de testes de unidade, deve ser poderoso o suficiente para detectar *bug's*, tornando ínfimo o tempo para o desenvolvedor efetuar a correção (FOWLER et. al., 2014). A seguir um exemplo de teste de unidade utilizando o framework JUnit 5:

Figura 8: Teste de unidade validando a classe *IcmsPadrao*

```java
package org.tcc.teste.unitario;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;
import org.tcc.classes.IcmsPadrao;

class IcmsPadraoTest {

    @Test
    void icmsPadrao_getValorImposto_ExpectedSuccess() {
        IcmsPadrao icms = new IcmsPadrao(18.00, 79.90);

        assertEquals(14.382, icms.getValorImposto());
    }

    @Test
    void icmsPadrao_construtor_ExpectedPercentualArgumentException() {
        assertThrows(IllegalArgumentException.class, () -> new IcmsPadrao(0.0, 79.90));
    }

    @Test
    void icmsPadrao_construtor_ExpectedValorBaseDeCalculoArgumentException() {
        assertThrows(IllegalArgumentException.class, () -> new IcmsPadrao(12.00, 0.00));
    }
}
```

Fonte: MACIEL, Juliano Ribeiro de Souza 2020.

Figura 9: Testes de unidade executados e aprovados



Fonte: MACIEL, Juliano Ribeiro de Souza 2020

De acordo com Khorikov (2020), um conjunto de testes bem-sucedido, deve estar integrado ao ciclo de desenvolvimento do produto, visar as partes mais importantes da base de código fonte e fornecer valor máximo com custos de manutenção mínimos. Já Osherove (2014), defende que uma boa base de testes unitários deve ser desenvolvida seguindo três princípios, sendo elas, confiabilidade, manutenção e legibilidade.

3.4. Refatoração

Software's são concebidos com o intuito de resolverem problemas de forma automatizada e eficaz, além de estarem em constante evolução, tendo novas funcionalidades adicionadas em seu portfólio, em suma, o *software* recebe informações às processa e gera resultados consolidados. Até aqui, apresentamos soluções de problemas comuns e como implementá los com alto grau de qualidade, porém nem sempre um desenvolvedor se deparará com um cenário inicial, onde poderá projetar a aplicação, neste contexto introduzimos a refatoração.

Fowler et. al. (2018) propõe que, refatoração é o processo de mudar um sistema de *software* de uma forma que não altera o seu comportamento externo do código, melhorando sua estrutura interna. É um forma disciplinada de limpar o código, que minimiza as chances de introdução de *bug's*. Em essência, ao refatorar, é melhorado o design do código após ter sido escrito. Já Shvets (2014), diz que refatoração é a busca sistemática pela melhoria do código, combatendo a dívida técnica e transformado o caos e desordem em código limpo de design simples.

> Dívida técnica é uma metáfora que se refere às consequências do desenvolvimento de software deficiente. Entregar código imaturo é como entrar em dívida. Um pouco de dívida agiliza o desenvolvimento, contanto que ela seja paga, prontamente com refatoração (Cunningham, 1992). Também pode ser definida como, uma lacuna entre o estado atual do software e algum estado hipotético 'ideal' no qual o sistema é otimamente bem sucedido em um ambiente em particular (Brown, 2010) ou qualquer parte do sistema atual que é considerado sub ótimo de uma perspectiva técnica (Ktata, 2010 - Tom, 2013). (SANTOS, C. R. 2015).

Podemos afirmar que dívida técnica se remete a soluções paliativas que implementamos em produtos de *software*, sem que apliquemos o mínimo de padrões de projeto ou boas práticas de programação, além de uma cobertura mínima de código por testes de unidade automatizados. Tais dívidas podem ser geradas por pressão do mercado para determinada funcionalidade no produto, pela falta de visão das consequências a médio e longo prazo do gestor do projeto ou por pura incompetência do desenvolvedor, todavia deverá ser paga, seja por meio de refatoração ou pela morte do produto, pois torna-se impossível sua manutenção e incremento de novas funcionalidades.

A refatoração deve ser feita como uma série de pequenas alterações, cada uma das quais torna o código existente um pouco melhor, mantendo as

funcionalidades atuais do *software* em funcionamento (SHVETS, 2014). A seguir será apresentado um exemplo de refatoração onde será extraído uma validação de uma subclasse para a superclasse além de adicionar uma nova validação para os atributos obrigatórios informados no construtor da classe, tornando-se padrão para futuras implementações.

Figura 10: Classe *AbstractImposto* pré refatoração

```java
package org.tcc.classes;

public abstract class AbstractImposto {
    private double percentualAliquota;
    private double valorBaseDeCalculo;

    public AbstractImposto(double percentualAliquota, double valorBaseDeCalculo) {
        this.percentualAliquota = percentualAliquota;
        this.valorBaseDeCalculo = valorBaseDeCalculo;
    }

    protected double getPercentualAliquota() {
        return this.percentualAliquota;
    }

    protected double getValorBaseDeCalculo() {
        return this.valorBaseDeCalculo;
    }

    public abstract double getValorImposto();
}
```

Fonte: MACIEL, Juliano Ribeiro de Souza 2020.

Figura 11: Classe IcmsPadrao pré refatoração

```java
package org.tcc.classes;

public class IcmsPadrao extends AbstractImposto{

    public IcmsPadrao(double percentualAliquota, double valorBaseDeCalculo) {
        super(percentualAliquota, valorBaseDeCalculo);

        if(this.getPercentualAliquota() <= 0)
            throw new IllegalArgumentException("Percentual Alíquota inválido! Verifique.");
        else if(this.getValorBaseDeCalculo() <= 0)
            throw new IllegalArgumentException("Valor Base de Cálculo inválido! Verifique.");
    }

    @Override
    public double getValorImposto() {
        return getValorBaseDeCalculo() * (this.getPercentualAliquota() / 100);
    }
}
```

Fonte: MACIEL, Juliano Ribeiro de Souza 2020.

Figura 12: Classe *IcmsComReducao* pré refatoração

```java
package org.tcc.classes;

public class IcmsComReducao extends AbstractImposto {

    private double percentualDeReducao;

    public IcmsComReducao(double percentualAliquota, double percentualReducao, double valorBaseCalculo) {
        super(percentualAliquota, valorBaseCalculo);
        this.percentualDeReducao = percentualReducao;

        if(this.getPercentualAliquota() <= 0)
            throw new IllegalArgumentException("Percentual Alíquota inválido! Verifique.");
        else if(this.getValorBaseDeCalculo() <= 0)
            throw new IllegalArgumentException("Valor Base de Cálculo inválido! Verifique.");
        else if(this.percentualDeReducao <= 0)
            throw new IllegalArgumentException("Percentual de redução inválido! Verifique.");
    }

    @Override
    public double getValorImposto() {
        return getValorBaseDeCalculo()
                * (1 - ((getPercentualAliquota() / 100) * (1 - this.percentualDeReducao)))
                / (1 - (getPercentualAliquota() / 100)) - getValorBaseDeCalculo();
    }
}
```

Fonte: MACIEL, Juliano Ribeiro de Souza 2020.

Analisando as (figuras 10, 11 e 12), é observado código duplicado para efetuar a validação dos parâmetros passados via construtor, a refatoração proposta é abstrair o que há em comum em ambas as classes para um método isolado para que a classe *IcmsComReducao* efetue sua particularidade e compartilhe dos recursos de sua superclasse, tendo assim aplicado polimorfismo entre os objetos posteriormente criados.

Figura 13: Classe *AbstractImposto* pós refatoração

```java
package org.tcc.classes;

public abstract class AbstractImposto {
    private double percentualAliquota;
    private double valorBaseDeCalculo;

    public AbstractImposto(double percentualAliquota, double valorBaseDeCalculo) {
        this.percentualAliquota = percentualAliquota;
        this.valorBaseDeCalculo = valorBaseDeCalculo;

        validarDadosImpostoTransacao();
    }

    protected void validarDadosImpostoTransacao() {
        if(this.getPercentualAliquota() <= 0)
            throw new IllegalArgumentException("Percentual Alíquota inválido! Verifique.");
        else if(this.getValorBaseDeCalculo() <= 0)
            throw new IllegalArgumentException("Valor Base de Cálculo inválido! Verifique.");
    }

    protected double getPercentualAliquota() {
        return this.percentualAliquota;
    }

    protected double getValorBaseDeCalculo() {
        return this.valorBaseDeCalculo;
    }

    public abstract double getValorImposto();
}
```

Fonte: MACIEL, Juliano Ribeiro de Souza 2020.

Figura 14: Classe *Icms* pós refatoração

```java
package org.tcc.classes;

public class IcmsPadrao extends AbstractImposto{

    public IcmsPadrao(double percentualAliquota, double valorBaseDeCalculo) {
        super(percentualAliquota, valorBaseDeCalculo);
    }

    @Override
    public double getValorImposto() {
        return getValorBaseDeCalculo() * (this.getPercentualAliquota() / 100);
    }
}
```

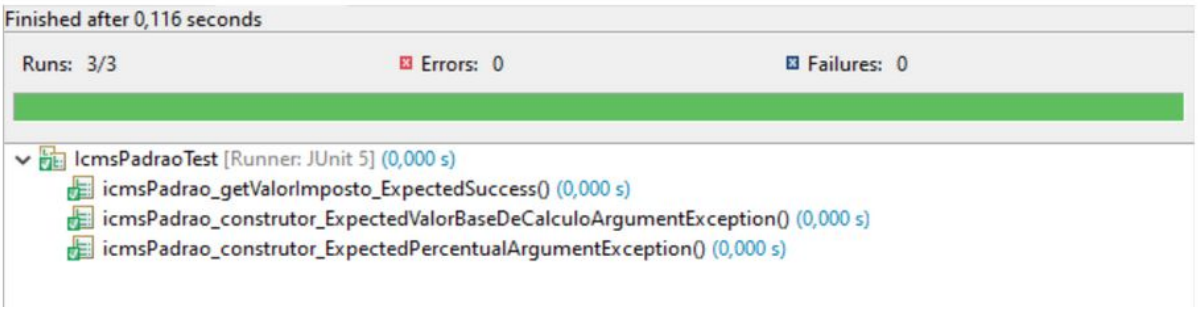
Fonte: MACIEL, Juliano Ribeiro de Souza 2020.

Figura 15: Classe *IcmsComReducao* pós refatoração

```java
package org.tcc.classes;

public class IcmsComReducao extends AbstractImposto {

    private double percentualDeReducao;

    public IcmsComReducao(double percentualAliquota, double percentualReducao, double valorBaseCalculo) {
        super(percentualAliquota, valorBaseCalculo);
        this.percentualDeReducao = percentualReducao;

        if(this.percentualDeReducao <= 0)
            throw new IllegalArgumentException("Percentual de redução inválido! Verifique.");
    }

    @Override
    public double getValorImposto() {
        return getValorBaseDeCalculo()
                * (1 - ((getPercentualAliquota() / 100) * (1 - this.percentualDeReducao)))
                / (1 - (getPercentualAliquota() / 100)) - getValorBaseDeCalculo();
    }
}
```

Fonte: MACIEL, Juliano Ribeiro de Souza 2020.

Figura 14: Testes unitários executados após refatoração das classes apresentadas nas (figuras 13, 14 e 15)



Fonte: MACIEL, Juliano Ribeiro de Souza 2020.

Após a refatoração não houve impacto no comportamento esperado do software, visto que os testes unitários garantiram a integridade da aplicação. Com a refatoração, podemos pegar um código ruim e caótico, e retrabalhá lo em um código bem estruturado. Cada etapa pode ser simples movendo um campo de uma classe para outra, extraindo um código para fora de um método e torná-lo seu próprio método, ou movendo código para cima ou para baixo em uma hierarquia. No entanto,

> o efeito cumulativo dessas pequenas mudanças melhoraram radicalmente o design do software. É o oposto exato da noção de decadência do software (FOWLER et al. 2018).

4. Relato

Dadas as estruturas de construção e manutenção de um produto de *software* apresentadas até aqui, tomando como base a experiência adquirida ao longo de dois anos atuando como desenvolvedor em um produto que não foi bem gerido pela equipe mantenedora, será relatada, a estratégia utilizada para aplicar todos os conceitos apresentados neste artigo, sendo assim renascendo um produto de *software* e torná lo peça fundamental na estratégia de expansão comercial do varejo de moda pela TOTVS S/A.

O primeiro passo levantado neste cenário críticos, foi assumir o caos instaurado, a adição de novas funcionalidades pode deixar problema ainda maior, sendo assim necessária uma pausa para traçar a estratégia a ser seguida. Mas por onde começar? Para reverter a situação, foi necessário ir a fundo, analisando a aberturas de tickets de *bug's* para identificar o que chamamos de causa raiz, ou seja, qual rotina que refatorada causará maior impacto positivo na usabilidade dos usuários, diminuindo abertura de tickets e manutenções de *bug's*. Tão logo, percebemos que o núcleo da aplicação, a funcionalidade de vendas, era a principal "dor para nossos clientes"[4], suporte e nos mesmos (desenvolvedores), identificada a causa raiz o próximo passo é a montagem do ciclo de implementações.

Após a identificação da "causa raiz"[5] mapear qual problema atacar primeiro é essencial, usando o mesmo conceito apresentado no primeiro passo, pensamos em qual alteração / refatoração na tela de vendas causaria maior impacto positivo para os usuários (foram distribuídas várias tarefas a serem executadas em paralelo pelo time desenvolvedores, porém citarei um em especifico para tomarmos como exemplo), logo foi um consenso a aplicação correta do padrão de projeto Data Transfer Object (Objeto de Transferência de Dados) ou DTO, em suma, a cada registro de produto adicionado a uma transação de venda tornava o carregamento das informações em tela mais lento, pelo design das entidades conterem muitos

[4] Um problema persistente ou recorrente que freqüentemente incomoda ou irrita os clientes.
[5] O Diagrama de Ishikawa, também conhecido como Diagrama de Causa e Efeito ou Diagrama Espinha de Peixe, é uma ferramenta gráfica utilizada para mapear-se as causas raiz do problema.

relacionamentos e a cada iteração do usuário toda essa carga de dados era reprocessada e retornada para apresentação em tela, o objeto levava muito tempo para ser serializado retornado para o cliente acionador da rotina e desserializado em tela para o usuário. O padrão DTO veio para suprimir esse problema, basicamente sua implementação se concentra em criar um objeto distinto que possui apenas os atributos necessários para serem retornados na chamada requisitada pelo cliente da aplicação.

Conseguimos encontrar uma refatoração de uma causa raiz que terá um impacto significativo na usabilidade dos usuários, agora como assegurar a qualidade da refatoração no código fonte, para não gerarmos mais um legado a ser mantido pelo time de desenvolvedores. No decorrer deste artigo apresentamos que a aplicação de testes unitários automatizados respalda aos integrantes da equipe independente do nível de senioridade, tornando intervenções no código fonte seguras, garantindo que funcionalidades não quebrem. Firmamos alguns acordos de cavalheiros, a partir de então todo pull-request criado para a branch principal do projeto seguiu algumas regras como, a implementação testes unitários automatizados e aplicação de boas práticas de nomenclatura, além de passar pelo crivo de ao menos três outros desenvolvedores sendo um deles de nível sênior. Este foi o início conforme tornou-se comum para a equipe e o nível do time aumentou, algumas regras foram removidas como a necessidade efetiva de um desenvolvedor senior aprovar um pull request, e novas regras foram incrementadas como seguir princípios de SOLID[6], sempre visando a qualidade e manutenibilidade do produto, sendo por aplicação de novos padrões de projeto, novos conceitos de boas práticas de programação ou *frameworks* de testes automatizados de carga e stress.

Em alguns meses de trabalho o produto foi estabilizado, utilizando também conceitos de estrangulamento[7] de *software* para que aos poucos rotinas legadas fossem desacopladas e posteriormente expurgadas do produto. A estabilização foi chave para retomada da confiança do produto pela gestão e areá comercial da empresa, proporcionando insumos para estratégias comerciais de expansão no

[6] SOLID é um acrônimo dos cinco primeiros princípios da programação orientada a objetos e design de código identificados por Robert Cecil Martin.
[7] Leitura complementar sobre o *Strangle Pattern Strategy*, pode ser encontrados no site *InfoQ.* Disponível em: <https://www.infoq.com/br/articles/strangle-Pattern-strategy/>.

varejo de moda. A reabilitação do produto no mercado, pesou consideravelmente pelo padrão de qualidade implementado, gerando valor agregado para mantenedores e clientes, tornando fácil a adição de novas funcionalidades e correções de *bug's*.

5. Considerações Finais

As opções disponíveis na literatura sobre padrões de projeto, arquitetura de *software* e boas práticas no desenvolvimento são amplas e atacam diversos aspectos relevantes em produtos de *software*. Neste trabalho foi aplicada uma fração das técnicas disponíveis a fim de reverter um cenário caótico na concepção e manutenção de um produto de *software*, pois pequenas mudanças podem trazer melhoras significativas a ele no médio / longo prazo, facilitando a incorporação de novos profissionais, bem como identificação de *bug's* por isolar responsabilidades, além de tornar possível a adição de novas funcionalidades sem incorrer em *bug's* naquelas já implementadas.

No cenário apresentado a confiança dos líderes em sua equipe é de suma importância, pois não é fácil uma quebra de paradigma: na indústria de *software* a pressão por resultados é imensa, mas organizar o processo fundamentar aspectos arquiteturais do produto de *software* garantirá maior probabilidade de sucesso do mesmo. Além disso a sintonia entre líder e liderados direciona o foco de modo a colher resultados satisfatórios no médio / longo prazo.

REFERÊNCIAS

CORNÉLIO, Márcio Lopes. **Refactorings as Formal Refinements**. Tese de Doutorado, RI UFPE, Recife, 2004. Disponível em: <https://repositorio.ufpe.br/bitstream/123456789/1891/1/arquivo4837_1.pdf>. Acesso em: 30/09/2020.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph et al. **Design Pattern: Elements of Reusable Object-Oriented Software**, Addison-Wesley, 1994.

LARMAN, Craig. **Applying UML and Patterns**, Prentice Hall, 2007.

MARTIN, Robert Cecil. **Clean Code - A Handbook of Agile Software Craftsmanship**, Prentice Hall, 2008.

FREEMAN, Eric; ROBSON, Elisabeth et al. **Head First: Design Patterns**, O'Reilly, 2014.

MCCONNELL Steve. **Code Complete: A Practical Handbook of Software Construction**, Microsoft Press, 2004.

ORACLE. **Code Conventions for the Java Programming Language**, 1999. Disponível em: <https://www.oracle.com/java/technologies/javase/codeconventions-contents.html>. Acesso em: 04/10/2020

SHVETS, Alexander. **What is Refactoring**, 2014. Disponível em: <https://refactoring.guru/refactoring/what-is-refactoring>. Acesso em: 26/09/2020

SHVETS, Alexander. **Factory Method**, 2014. Disponível em: <https://refactoring.guru/design-patterns/factory-method>. Acesso em: 29/09/2020

COPLIEN, James O'connell. **Software Patterns**, New York NY, SIGS Books & Multimedia, 1996. Disponivel em: <https://csis.pace.edu/~bergin/dcs/SoftwarePatterns_Coplien.pdf>. Acesso em: 10/10/2020.

FOWLER, Martin; BECK, Kent et al.. **Refactoring: Improving the Design of Existing Code**, Addison-Wesley Professional, 2018.

SANTOS, Ciro Goulart. **Um Estudo Empírico Sobre a Gerência de Dívida Técnica em Projetos de Desenvolvimento de Software que Utilizam Scrum.** Dissertação de Mestrado, PUCRS, Porto Alegre, 2015.

BROWN, Nanette; CAI, Yuanfang; GUO, Yueou et al.. **Managing Technical Debt in Software-Reliant Systems**, Conference: Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Santa Fe, NM, USA, 2010.

KTATA, Oualid; LÉVESQUE, Ghislain et al.. **Designing and implementing a measurement program for scrum teams: what do agile developers really need and want?**, Conference: Canadian Conference on Computer Science & Software Engineering, C3S2E 2010, Montreal, Quebec, Canada, 2010.

TOM, Edith; AURUM, Aybuke; VIDGEN, Richard et al.. **An exploration of technical debt, Journal of Systems and Software**, Journal of Systems and Software, vol. 86-6, 2013. Disponível em: <https://www.researchgate.net/publication/256991988_An_exploration_of_technical_debt>. Acesso em: 16/10/2020.

KHORIKOV, Vladimir. **Unit Testing - Principles, Practices and Patterns**, Manning Publications, 2020.

OSHEROVE, Roy. **The Art of Unit Testing**, Mitp, 2014.