PONTIFÍCIA UNIVERSIDADE CATÓLICA DE SÃO PAULO
COGEAE – PUC/SP
ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

RICARDO JOAQUIM MUNIZ JUNIOR

CÓDIGO-FONTE: MELHORIA DE LEGIBILIDADE POR FLUENT INTEFACE

SÃO PAULO
2013

RICARDO JOAQUIM MUNIZ JUNIOR

CÓDIGO-FONTE: MELHORIA DE LEGIBILIDADE POR FLUENT INTERFACE

Monografia apresentada ao Curso de Especialização em Engenharia de Software, da Pontifícia Universidade Católica de São Paulo – COGEAE, como pré-requisito para obtenção do título de Especialista em Engenharia de Software.

Área de concentração: Engenharia de Software.

Orientador: Prof. Dr. Ítalo Santiago Vega

SÃO PAULO

2013

RICARDO JOAQUIM MUNIZ JUNIOR

CÓDIGO-FONTE: MELHORIA DE LEGIBILIDADE POR FLUENT INTERFACE

Monografia de Conclusão de Curso apresentada à Pontifícia Universidade Católica de São Paulo (PUC/SP), como parte dos requisitos para obtenção do título especialista em Engenharia de Software sob a orientação do Prof. Dr. Ítalo Santiago Vega.

Dedico este trabalho a Deus aos meus pais, a minha esposa, ao meu enteado e aos amigos que diretamente ou indiretamente me auxiliaram na realização deste trabalho.

Ricardo Joaquim Muniz Junior

AGRADECIMENTOS

Quero agradecer, em primeiro lugar, a Deus, pela força e coragem durante toda esta longa caminhada;

Aos meus pais Ricardo e Maria Elsa, que me incentivaram a continuar meus estudos e me aprofundar cada vez mais;

À minha esposa Simone e enteado Lucas que de forma especial e carinhosa me deram força e coragem, me apoiando nos momentos de dificuldades;

Agradeço também a todos os amigos de profissão e de estudos, que mesmo sem saberem, contribuíram para minha vida profissional e acadêmica;

E a todos que colaboraram de forma direta e indireta na realização deste trabalho.

Ao Prof. Dr. Ítalo Santiago Vega, pela paciência, pela perseverança, pelas orientações e principalmente pelas revisões deste trabalho, dedico minha admiração e respeito.

Ricardo Joaquim Muniz Junior

Sei que o meu trabalho é uma gota no oceano, mas sem ele, o oceano seria menor.

(Madre Teresa de Calcutá)

RESUMO

O resultado desta pesquisa é uma demonstração de aplicação da técnica *Fluent Interface*, que é utilizada no desenvolvimento de DSLs Internas, técnica esta, que foi cunhada por Martin Fowler e Eric Evans. Ela altera os métodos *Setters*, que são métodos responsáveis pela atribuição de valor das propriedades dos objetos, para que os mesmos retornem a própria instância do objeto, podendo encadear múltiplas chamadas de um mesmo objeto. Será utilizada a técnica para melhorar a legibilidade do código-fonte, sendo que, ao aumentar a legibilidade de um software, aumenta também a manutenibilidade do mesmo, tornando-o mais fácil de ser adaptado, corrigido e/ou evoluído. A técnica é aplicada em um código-fonte, e após a sua aplicação compara-se o código, antes e depois, analisando se houve melhoria na sua legibilidade, ou não. São apresentadas no trabalho as definições de: DSLs, DSLs internas, da técnica *Fluent Interface* e legibilidade

PALAVRAS-CHAVE: DSLs Internas, *Fluent Interface*, legibilidade.

ABSTRACT

The result of this research is a demonstration of application of the technique Fluent Interface, which is used in developing Internal DSLs, technique which was coined by Martin Fowler and Eric Evans. It alters Setters methods, methods that are responsible for allocating the value of the properties of objects, so that they return the object own instance, being able to chaining multiple calls of the same object. The technique will be used to improve the readability of source code, and, to increase the readability of software, increasing the maintainability of the same, making it easier to be adapted, adjusted and / or evolved. The technique is applied to a source-code, and after application compares the code before and after, analyzing whether there was an improvement in readability, or not. The definitions presented in the work: DSLs, internal DSLs, technical Fluent Interface and readability.

KEYWORDS: Internal DSLs, Fluent Interface, readability.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API **-** Application Program Interface

DSL **-** Domain-specific language

IDE - Integrated Developmente Environment

LINQ **-** Language Integrated Query

SUMÁRIO

1 INTRODUÇÃO

A técnica *Fluent Interface* é comumente associada a DSLs *(Domain-Specific Language)* internas, é um tipo de DSL. Constantemente se faz uso delas, escrevendo estilos com *CSS (Cascading Style Sheets)*, utilizando o *Ruby on Rails*, ou usando *expressões regulares*.

Neste trabalho é aplicada a técnica *Fluent Interface* em um trecho de código, a fim de exemplificar o uso da mesma, e será analisado se o código ficará mais legível após a aplicação da técnica, ou seja, se o código ficou mais fluído e menos redundante. O projeto se diferencia pelo fato que abordará o tema DSL e a técnica *Fluent Interface* com o contexto da legibilidade.

“A legibilidade afeta a confiabilidade tanto nas fases de escrita como nas de manutenção no ciclo de vida. Programas de difícil leitura complicam também sua escrita e a sua modificação” (Sebesta, 2000). A legibilidade do código-fonte é importante, pois aumentando-a deixa o código mais fácil de ser lido e compreendido pela pessoa que o lê, com isso aumentar a modificabilidade do software, que o torna mais manutenível.

Por ser prática e simples de ser aplicada, comparada a técnicas que precisam modificam muito a arquitetura do software, essa técnica pode ser utilizada em projetos que estão iniciando, bem como os já iniciados.

O presente trabalho é destinado a desenvolvedores, arquitetos e engenheiros de software, bem como pessoas interessadas em engenharia de software e DSLs, que estão passando por problemas de legibilidade, dificuldade de manutenção, alta taxa de erros, entre outros.

1.1 Necessidade

DSLs são desenvolvidas para domínios específicos e por esse motivo facilitam a comunicação dos desenvolvedores e analistas de negócios. Sua utilização é ampla, existindo diversas DSLs de sucesso, como: *CSS*, *Ruby on Rails*, *Expressão Regulares* entre outras.

Nesses últimos anos bibliografias foram lançadas sobre o assunto, entre elas: “DSL - Linguagens Específicas de Domínio” (Fowler, 2013) e “DSLs in action” (Ghosh,

2011), ambas utilizadas neste trabalho, que por abordarem com amplitude o tema tratado, motivaram o aprofundamento no assunto.

A técnica *Fluent Interface*, que é uma das técnicas utilizadas para na criação de DSLs internas e com o seu uso, torna a DSL mais simples e legível. Tendo em vista esse fato, surge a indagação sobre a necessidade de aplicar essa técnica no desenvolvimento de softwares, o que poderia tornar o código-fonte mais legível.

1.2 Objetivo

O objetivo deste trabalho de pesquisa é demonstrar e discutir a aplicação da técnica *Fluent Interface* em um código-fonte, verificando se com a aplicação da técnica houve melhoria na legibilidade do mesmo.

1.3 Revisão da Literatura

A legibilidade é um item importante na qualidade de software e exerce influência na sua manutenção, conforme os autores Permula, Jetti e Sajjala (2011): o fator crítico na manutenção da qualidade de software é a legibilidade. E a legibilidade de um programa está relacionada à sua manutenção, pois para realizar a manutenção de um software, é necessário ler e entender como o mesmo funciona para depois alterá-lo. Logo, se faz necessário desenvolver um software manutenível. Tornar o código mais legível, é torná-lo mais simples de ser lido e mais entendível (de assimilação mais simples).

1.4 Metodologia

Neste trabalho de pesquisa serão utilizados os métodos de procedimento comparativo e indutivo, com base em dados bibliográficos. Um levantamento bibliográfico sobre DSLs, DSLs internas, *Fluent Interface* e legibilidade compõem esses dados. Será aplicada a técnica proposta a um trecho de código e analisado se o mesmo ficará mais legível.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capitulo, irá ser aprofundado o conhecimento em assuntos que servirá de base para o resto do trabalho.

Primeiramente, será discutido sobre o significado de DSL, suas definições, quais são os tipos que existem. Depois, o assunto é aprofundado nas DSL Internas, que são um tipo de DSL. Após as DSL Internas, é a vez da técnica *Fluent Interface*, técnica utilizada nas construções de DSL Internas. Por fim, uma discussão sobre legibilidade.

2.1 DSL

DSL (*Domain-Specific Language* ou Linguagem de Domínio Específico, daqui em diante referida somente por DSL) é uma linguagem de programação que foca em um determinado domínio e a tempos vem ajudando desenvolvedores e especialistas de negócios; sendo auxiliando, agilizando ou facilitando o desenvolvimento em domínios específicos.

Ghoshi (2011), definiu DSL como: “Uma Linguagem de domínio específico é uma linguagem de programação que é focada a um problema específico [...] contém a sintaxe, semântica e conceitos do modelo no mesmo nível de abstração que o problema do domínio oferece” (Ghosh, 2011).

A respeito de linguagem de programação, Fowler (2013) definiu: “é usada por humanos a fim de instruir um computador a fazer algo” (Fowler, 2013), com a linguagem de programação, podemos combinar instruções a fim de realizarmos uma tarefa, que serão traduzidos em funções que podem ser executadas em computadores. Fowler (2013) dividiu as linguagens de programação em duas: linguagem de propósito geral e linguagem de domínio específico. Sendo linguagem de propósito geral a linguagem de programação que pode ser utilizada em diversos domínios e propósitos, como C#, C++, Java entre outras. Já a DSL é focada em um domínio específico, como foi visto na definição acima.

Fowler, dividiu DSL em 3 (três) tipos, DSL externas e DSL internas e bancadas de linguagem, definindo-as:

- DSL Externas, como sendo: “uma linguagem separada da linguagem principal da aplicação com a qual ela trabalha” e ainda “normalmente, uma DSL externa possui sua sintaxe customizada, mas usar a sintaxe de outra linguagem também é comum (*XML* é uma escolha frequente)” (Fowler, 2013). Nessa definição atenta-se que, ela é separada e diferente da linguagem da aplicação, ao desenvolver uma DSL em C#, a linguagem a ser utilizada na DSL será diferente da linguagem C#, e utiliza-se um interpretador customizado para ler a mesma.
- DSL Interna, como sendo: “uma maneira específica de usar uma linguagem de propósito geral. Um Script em uma DSL interna é um código válido em sua linguagem de propósito geral” e ainda “o resultado deve ter a cara de uma linguagem customizada, e não a cara de uma linguagem hospedeira” (Fowler, 2013). A diferença é que a DSL interna utiliza-se da linguagem de programação (linguagem de propósito geral) que é utilizada na aplicação, também conhecida como linguagem hospedeira, para executar o código, diferente das DSLs externas.
- Bancada de linguagem, como sendo: “é uma *IDE* especializada em definir e construir DSLs” e ainda “uma bancada de linguagem é usada não apenas para determinar a estrutura de uma DSL, mas também como um ambiente customizado de edição para que as pessoas escrevam *scripts* em DSLs” (Fowler, 2013), com as bancadas de linguagem se tem um ambiente focado no domínio, que é a *IDE,* a mesma possui um ambiente gráfico possibilitando modelar o domínio de forma visual, e assim facilitando a criação dos *scripts* DSL.

2.2 DSLs Internas

Como foi visto anteriormente, a DSL é dividida em três tipos, neste tópico irá ser discutido as DSL Internas.

“Uma DSL interna utiliza a infraestrutura de uma linguagem de programação existente (também chamado como: língua de acolhimento da DSL) para a construção semântica de domínio específico em cima dela” (Ghosh, 2011).

“Uma das DSLs internas mais populares, utilizadas atualmente é o *Rails*, implementado utilizando a linguagem de programação *Ruby*” (Ghosh, 2011) e cita ainda que: “quando você escreve código *Rails*, você está programando em Ruby, com base na semântica que o Rails implementa para o desenvolvimento aplicações web” (Ghosh, 2011). Por Rails, o autor cita o *framework Ruby on Rails*.

“Na maioria dos casos, uma DSL interna é implementada como uma biblioteca *top* da linguagem *host* existente” (Ghosh, 2011). Um ponto relevante é, DSL interna é implementada como uma biblioteca de software (biblioteca é um conjunto de métodos de programas/rotinas), que o desenvolvedor utiliza-se para criar a sua aplicação.

DSL interna é uma forma de usar uma linguagem de propósito geral em prol de um domínio específico, o que facilita a comunicação do desenvolvedor com o analista de negócios. Para auxiliar o desenvolvimento de uma DSL interna existem algumas técnicas que auxiliam o desenvolvimento, resolvendo problemas já conhecidos, como:

- Fechos aninhados;
- Sequência de funções;
- Funções aninhadas;
- Fluent Interface, utilizada neste trabalho.

2.3 Técnica Fluent Interface

Como foi visto, uma das técnica que auxiliam na criação de uma DSL Interna é a *Fluent Interface*. A técnica *Fluent Interface* (Interface fluente) é um termo cunhado por Eric Evans e Martin Fowler em meados de 2005. *Fluent Interface* faz uso do padrão encadeamento de métodos (*Method Chaining*), o que permite múltiplas chamadas, possibilitando ao desenvolvedor encadear métodos, a fim de que o código tenha mais fluidez e legibilidade, facilitando a leitura do mesmo.

Conforme Fowler (2013), encadeamento de métodos é fazer os métodos modificadores retornarem o objeto hospedeiro (objeto que recebe a mensagem), de forma que múltiplos modificadores possam ser invocados em uma única expressão. No encadeamento de métodos, os métodos de atribuição retornam o seu próprio objeto em cada chamada, podendo ter o seu estado interno modificado ou não. Ele muda a forma de se pensar em uma *API*, pois se difere do conceito: separação comando-consulta (*Command-query Separation*, termo proposto por Bertrand Meyer), que Fowler (2013) explica: “separação comando-consulta diz que diversos métodos em um objeto devem ser divididos em comandos e em consultas” e ainda “uma consulta é um método que retorna um valor, mas não modifica o estado observável do sistema. Um comando pode modificar o estado observável, mas não deve retornar um valor” (Fowler, 2013). Porém quando estamos usando o encadeamento de

métodos, devemos alterar os métodos *getters* e *setters* das propriedades do objeto, quebrando assim a separação comando-consulta, pois, alteramos o nosso método *setter* (que não tem retorno) para que ele retorne o próprio objeto depois de alterar o valor da propriedade.

Um ponto importante a salientar é que a técnica tende a aumentar a complexidade das classes para quem não tem familiaridade com a mesma, existindo assim a necessidade de um aprendizado, isto pode ser considerado como um ponto negativo.

Um exemplo de Fluent Interface / encadeamento de métodos é o *Linq* do Framework Microsoft, vide Figura 1.

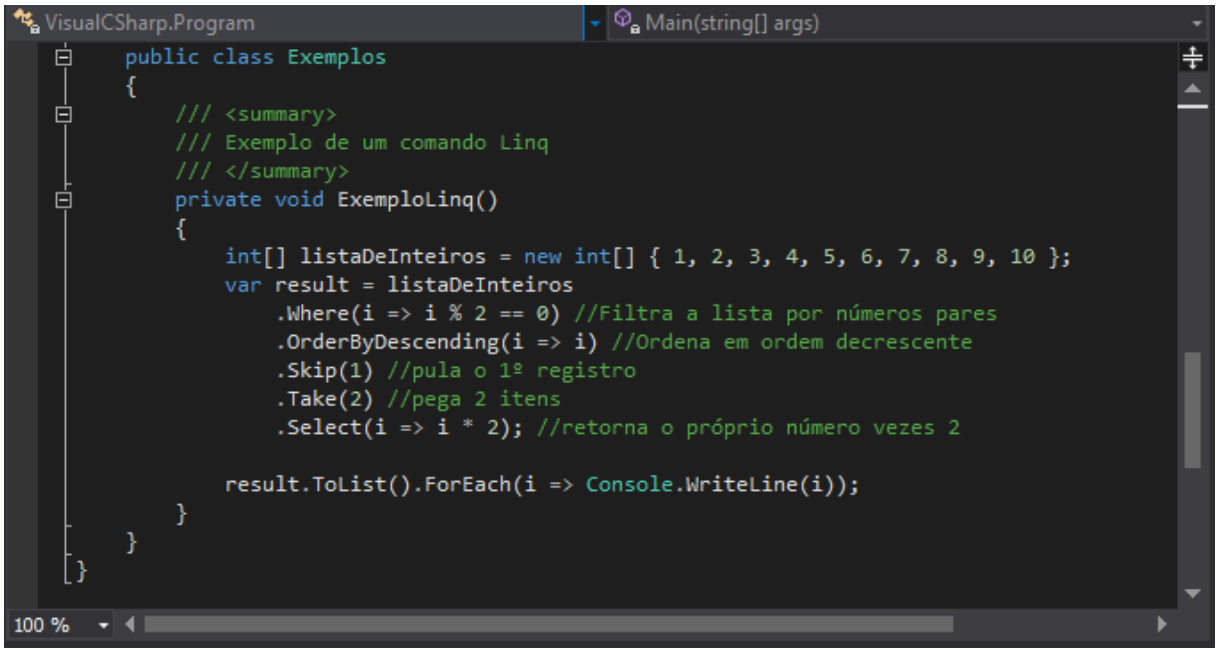

Figura 1 - Exemplo de um consulta Linq

2.4 Legibilidade do código fonte

Um problema que a técnica vista acima, a *Fluent Interface* se propõem a resolver, é a legibilidade do código-fonte, neste tópico será apresentado algumas definições sobre o assunto. Pressman define legibilidade através da seguinte afirmação: “a facilidade com que um software pode ser entendido, corrigido e/ou aumentado” (Pressman, 1995). De encontro à Pressman, Sebesta definiu em 2000 legibilidade como: “é a facilidade com que os programas podem ser lidos e entendidos” (Sebesta, 2000). De acordo com as definições, podemos observar que ambos definem a legibilidade como sendo, a facilidade para o entendimento do software.

Um fator que afeta a legibilidade conforme Sebesta (2000) é a capacidade de escrita (*Writability*), que é colocada como: “uma medida de quão facilmente uma linguagem pode ser usada para criar programas para um domínio de problema escolhido” e ainda, “a maioria das características da linguagem que afeta a legibilidade também afeta a capacidade de escrita”, e, “isso se segue diretamente do fato de que escrever um programa exige uma releitura frequentemente da parte que já foi escrita pelo programador” (Sebesta, 2000). O conhecimento do autor no domínio do problema bem como a fluência na linguagem utilizada, devem ser articulados de maneira que propiciem a legibilidade, o que exige uma releitura constante do código por parte de quem está escrevendo.

Conforme Permula, Jetti e Sajjala (2011), o fator crítico na manutenção da qualidade de software é a legibilidade. Os autores citam “qualidade de software”, que é definida por Pressman, como: “Conformidade a requisitos funcionais e de desempenho explicitamente declarados, a padrões de desenvolvimento claramente documentados e as características implícitas que são esperadas de todo software [...]” (Pressman, 1995).

“A legibilidade afeta a confiabilidade tanto nas fases de escrita como nas de manutenção no ciclo de vida. Programas de difícil leitura complicam também sua escrita e a sua modificação” (Sebesta, 2000).

“Uma vez que a manutenção frequentemente é feita por pessoas que não o autor original do software, uma legibilidade ruim pode tornar a tarefa extremamente desafiadora” (Sebesta, 2000).

De acordo com Sebesta, nem sempre quem escreve o código irá fazer a manutenção do mesmo. Por esse motivo, entende-se que se faz necessário que o autor do código, desenvolva-o de maneira clara e objetiva, procurando facilitar ao leitor a interpretação do mesmo. E, quanto menor for a complexidade do modelo computacional adotado e mais legível estiver o software, compreende-se que: mais entendível e manutenível o mesmo estará.

Alguns pesquisadores observaram que o ato de leitura de código é o componente mais demorado de todas as atividades de manutenção (Permula, Jetti, & Sajjala, 2011). De fato, isto ocorre, porque, na manutenção, quando um desenvolvedor corrige um código, bem como adicionar novas funcionalidades, se faz necessário lê-lo e compreendê-lo, para que seja possível alterá-lo. Quanto mais

complexo e menos legível for a arquitetura do software, mais difícil a sua compreensão e mais demorado o seu entendimento.

Pressman definiu manutenibilidade como: “O esforço dirigido para localizar e reparar erros num programa” (Pressman, 1995), e complementou dizendo que, a manutenibilidade tem relação com a simplicidade, e definiu simplicidade como: “o quanto um programa pode ser entendido sem dificuldade” (Pressman, 1995). O rápido entendimento de um software, está relacionado com a velocidade do entendimento do mesmo e da complexidade do software, sendo que um software simples, terá um entendimento mais fácil, do que um complexo.

3 RESULTADOS

Para ilustrar a aplicação da técnica em um código-fonte, será usada a classe Pessoa, que contém propriedades comuns a “pessoas”. Esta classe foi escolhida por ter propriedades simples e condizentes com os atributos de qualquer pessoa do mundo real, porém, a técnica pode ser aplicada em quaisquer classes que tenham propriedades, ou métodos que possam ser encadeados, ou seja, qualquer método que possa retornar a sua própria instância.

Será utilizado um trecho de código que contém a definição da classe de exemplo e no decorrer deste capítulo irá ser aplicado a técnica *Fluent Interface* na mesma. No exemplo da Tabela 1 pode-se observar como é feita a instanciação de um objeto da classe Pessoa, antes da técnica ser aplicada.

```
Pessoa p1 = new Pessoa();
   p1.Nome = "Fulano de Tal";
   p1.Casado = false;
   p1.Sexo = Pessoa.SexoEnum.Masculino;
   p1.TemFilhos = false;
   p1.Peso = 70;
   p1.Altura = 1.95;

Pessoa p2 = new Pessoa();
   p2.Nome = "Ciclana de Tal";
   p2.Casado = true;
   p2.Sexo = Pessoa.SexoEnum.Feminino;
   p2.TemFilhos = true;
   p2.Peso = 89;
   p2.Altura = 1.74;
```

Tabela 1 – Exemplo de código (sem a técnica Fluent Interface).

Abaixo está a definição da classe Pessoa.

```
public class Pessoa
{
   public enum SexoEnum { Masculino, Feminino }

   private string nome;
   public string Nome
   {
      get { return nome; }
      set { nome = value; }
   }

   private SexoEnum sexo;
```

```
    public SexoEnum Sexo
    {
        get { return sexo; }
        set { sexo = value; }
    }

    private bool temFilhos;
    public bool TemFilhos
    {
        get { return temFilhos; }
        set { temFilhos = value; }
    }

    private bool casado;
    public bool Casado
    {
        get { return casado; }
        set { casado = value; }
    }


    private double peso;
    public double Peso
    {
        get { return peso; }
        set { peso = value; }
    }

    private double altura;
    public double Altura
    {
        get { return altura; }
        set { altura = value; }
    }
}
```

Tabela 2 - Definição da Classe Pessoa (Sem a técnica Fluent Interface aplicada).

A técnica *Fluent Interface* consiste em substituir os métodos que alteram os valores das propriedades (*setters*) de um objeto, e fazer com que o mesmo realize a substituição do valor da propriedade e retorne para quem o chame, a própria instância. O código abaixo tem uma propriedade que contém o *setter* e o *getter*, neste exemplo é utilizado a linguagem da Microsoft, o *Visual C#*, o mesmo possui uma notação diferenciada, que contém os métodos *setter* e *getter* encapsulados dentro da propriedade.

```
public class Pessoa
{
    ...
```

```
    private string nome;
    public string Nome
    {
        get { return nome; } //Getter
        set { nome = value; } //Setter
    }
    ...
}
```

Tabela 3 - Exemplo de propriedade com Getter e Setter.

Aplicando a técnica muda-se a forma que o valor da propriedade é atribuída, como no exemplo abaixo, que foi alterada a propriedade “Nome”. Substitui-se a propriedade “Nome” por um método, que receberá um novo valor para “Nome” como parâmetro e retornará a própria instância de Pessoa.

```
public class Pessoa
{
    ...
    private string nome;
    public Pessoa Nome(string novoNome)
    {
        this.nome = novoNome; //Atribui o novo valor ao nome
        return this; //retorna a própria instância
    }
    ...
}
```

Tabela 4 - Exemplo da propriedade preparada para ser acessado por Fluent Interface.

Ao mudar o método de atribuição da propriedade, pode-se atribuir o nome da Pessoa por Fluent Interface. Na Figura 2 (abaixo), observa-se a instanciação do objeto Pessoa e a atribuição do valor “João da Silva” à propriedade “Nome” do objeto “p” (Pessoa).

```
Pessoa p = new Pessoa();
p.Nome = "João da Silva";
```

Figura 2 - Instanciação do objeto Pessoa sem Fluent Interface

Aplicando-se a técnica *Fluent Interface*, modifica-se a instanciação e atribuição de valor do objeto Pessoa, como no exemplo da Figura 3.

```
Pessoa p = new Pessoa().Nome("João da Silva");
```

Figura 3 - Instanciação do objeto Pessoa com Fluent Interface na mesma linha

Utilizando-se de características da linguagem de programação, poder-se-ia deixar a instanciação e atribuição à propriedade na mesma linha, ou na linha abaixo, conforme Figura 4:

```
Pessoa p = new Pessoa()
    .Nome("João da Silva");
```

Figura 4 - Instanciação do objeto Pessoa com Fluent Interface em outra linha

Aplicando-se a técnica *Fluent Interface* ao exemplo utilizado na Tabela 2, os resultados apresentados serão conforme segue o exemplo na Tabela 5.

```
Pessoa p1 = new Pessoa()
    .Nome("Fulano de Tal")
    .Masculino()
    .Peso(70)
    .Altura(1.95);

Pessoa p2 = new Pessoa()
    .Nome("Ciclana de Tal")
    .Casado()
    .Feminino()
    .TemFilhos()
    .Peso(89)
    .Altura(1.74);
```

Tabela 5 - Exemplo de Código (utilizando a técnica Fluent Interface).

No exemplo da Tabela 5, foi optado, por quebrar o código em linhas para facilitar a leitura, mas, poder-se-ia implementar na mesma linha, conforme Figura 5.

```
Pessoa p1 = new Pessoa().Nome("Fulano de Tal").Masculino().Peso(70).Altura(1.95);
```

Figura 5 - Instanciação do objeto pessoa com atribuições complexas

Na Tabela 6, observa-se a definição da classe Pessoa, com a técnica *Fluent Interface* aplicada.

```
public class Pessoa
{
    public enum SexoEnum { Masculino, Feminino }

    private string nome;
    public Pessoa Nome(string novoNome)
    {
        this.nome = novoNome;
        return this;
    }

    private SexoEnum sexo;
    public Pessoa Feminino()
    {
        this.sexo = SexoEnum.Feminino;
        return this;
    }
    public Pessoa Masculino()
    {
        this.sexo = SexoEnum.Masculino;
        return this;
    }

    private bool temFilhos;
    public Pessoa TemFilhos()
    {
        this.temFilhos = true;
        return this;
    }

    private bool casado;
    public Pessoa Casado()
    {
        this.casado = true;
        return this;
    }


    private double peso;
    public Pessoa Peso(double novoPeso)
    {
        this.peso = novoPeso;
        return this;
```

```
    }

    private double altura;
    public Pessoa Altura(double novaAltura)
    {
        this.altura = novaAltura;
        return this;
    }
}
```

Tabela 6 - Definição da Classe Pessoa com a ténica Fluent Interface aplicada.

Pode-se observar o código da Tabela 6 a técnica *Fluent Interface* aplicada na classe de exemplo “Pessoa”, neste ponto já é possível instanciá-la conforme feito na Tabela 5.

Portanto, a definição da classe “Pessoa” demonstrada na Tabela 2, depois da técnica aplicada pode ser vista na Tabela 6. Alterada a definição da classe Pessoa, a instanciação demonstrada no código da Tabela 1, passa a ser feita como o código da Tabela 5. Atingindo assim o objetivo de demonstrar um exemplo de código-fonte antes e depois da técnica ser aplicada.

4 CONSIDERAÇÕES FINAIS

A aplicação da técnica foi representada e demostrada no exemplo do capítulo 3, utilizando-se uma instância do objeto Pessoa e atribuições as suas propriedades. Foi utilizada uma classe com estrutura simples para facilitar o entendimento (entende-se por estrutura simples que os setters do exemplo não fazem cálculo nenhum, somente alteram o valor interno das propriedades), porém a técnica pode ser aplicada também em classes complexas. O objetivo é trazer sentido fluído e legibilidade aos códigos, como já ocorre quando aplicada nas DSLs.

Antes da técnica ser aplicada o código era repetitivo, pois existia o nome da instância em cada atribuição de valor o que dificultava a sua leitura e fluidez.

Como observa-se na Tabela 6, que traz o exemplo do código alterado, com a aplicação da técnica Fluent Interface, constata-se que ele ficou menos repetitivo, com um sentido mais fluido e legível, pois foram encadeadas as atribuições de valores, possibilitando uma leitura facilitada.

Sugere-se que com a técnica aplicada ao código (Tabela 6), até um leitor que esteja familiarizado com a lógica de programação, mas que não conhece o código em questão, poderia lê-lo da seguinte forma: “nova pessoa ‘p1’, com nome de Fulano de Tal, masculino, com peso de 70, e com altura de 1.95”.

Em contraposição à análise feita em relação ao código da Tabela 6, temos o código da Tabela 1 (antes da técnica aplicada), onde, devido a repetição e falta de fluidez não se tem a mesma facilidade de leitura. A leitura do código da Tabela 1, ocorre-se da seguinte maneira: “Nova Pessoa ‘p1’, p1 nome Fulano de tal, p1 Casado falso, p1 sexo masculino, p1 tem filhos false, p1 peso 70, p1 altura 1.95”.

Assim como é possível observar, o código da Tabela 6, o qual aplicou-se a técnica Fluent Interface, ficou mais legível, em relação ao código da Tabela 1, confirmando a suspeita inicial que, ao aplicar a técnica Fluent Interface no código-fonte de exemplo, o mesmo se tornou mais legível.

5 TRABALHOS FUTUROS

Como possíveis trabalhos futuros, pode-se apontar:

- Analisar o Tempo de desenvolvimento x Legibilidade, quando aplica-se a técnica *Fluent Interface*;
- Verificar e levantar se existem pontos negativos da utilização da técnica *Fluent Interface*;
- Verificar se outras técnicas utilizadas em DSL internas, também ajudam a melhorar a legibilidade.

REFERÊNCIAS

FOWLER, Martin. **Patterns of Enterprise Application Architecture**. Boston: Addison-Wesley Professional, 2002.

FOWLER, Martin. **DSL: linguagens específicas de domínio**. Tradução Eduardo Kessler Piveta. Porto Alegre: Bookman, 2013.

GHOSH, Debasish. **DSLs in action**. Stamford: Maning, 2011.

PERMULA, Rambabu; JETTI, Kumar; SAJJALA, Praneeth; IEEE Std:Framework to measure and maintain the quality of software using the concept of Code Readability. **International Journal of Computer Technology and Applications**, cidade, v. 2, n 5, p. 1466-1471, out. 2011. ISSN: 2229-6093.

PRESSMAN, Roger S. **Engenharia de Software**. Tradução José Carlos Barbosa dos Santos. São Paulo: Makron Books, 1995.

SEBESTA, Robert W. **Conceitos de Linguagens de Programação**. Porto Alegre: Bookman, 2000.

GLOSSÁRIO

API: Application Programming Interface ou Interface de Programação de Aplicativos é um conjunto de objetos e métodos criados para que outros aplicativos façam uso, sem se envolver com a forma que foram implementados.

APIs: veja *API*.

C#: Veja *Visual C#.*

CSS: Cascading Style Sheets é uma linguagem de estilo usada para definir estilos em um documento *HTML*.

DSL: Domain-Specific Language ou Linguagem de Domínio Específico,é uma linguagem de programação que é focada a um problema específico.

Encadeamento de Métodos: ou method chaining, é uma técnica que permite que se faça várias múltiplas chamadas métodos.

Expressões Regulares: permite escrever expressões em cadeia de caracteres para achar correspondências em textos.

Fluent Interface: utiliza-se da técnica Encadeamento de Métodos para dar sentido um sentido mais fluído ao código.

Getter: É o método público responsável por fazer a leitura e devolver o valor de uma variável interna.

Getters Veja *Getter*.

HTML: Hypertext Markup Language ou linguagem de marcação de hipertexto, é uma linguagem de marcação utilizada para criar documentos na web.

IDE: (Integrated Development Environment), um ambiente integrado para desenvolvimento de software, atualmente temos o Visual Studio da Microsoft para

desenvolvimento com a linguagem com .Net; Eclipse da IBM e NetBeans da Oracle, ambos para desenvolvimento em Java.

Rails: Veja Ruby on Rails.

Ruby: Linguagem de programação dinâmica, *open source*.

Ruby on Rails: Framework desenvolvido em *Ruby*.

Setter: É o método público responsável pela alteração de uma variável interna.

Setters: Veja *Setter*.

Visual C#: Linguagem de programação orientada à objetos, da Microsoft.

XML: (Extensible Markup Language) é uma linguagem de marcação.