UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DEPARTAMENTO ACADÊMICO DE INFORMÁTICA

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

VÍCTOR PEDROSO AMBIEL BARROS

UM MÉTODO PARA A REFATORAÇÃO DE SOFTWARE BASEADO EM FRAMEWORKS DE DOMÍNIO

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA

2015

VÍCTOR PEDROSO AMBIEL BARROS

UM MÉTODO PARA A REFATORAÇÃO DE SOFTWARE BASEADO EM FRAMEWORKS DE DOMÍNIO

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação, do Departamento Acadêmico de Informática, da Universidade Tecnológica Federal do Paraná.

Orientadora: Profª. Dr. Simone Nasser Matos

PONTA GROSSA

2015

Ministério da Educação

Universidade Tecnológica Federal do Paraná

Câmpus Ponta Grossa

Diretoria de Graduação e Educação Profissional

TERMO DE APROVAÇÃO

Um Método para a Refatoração de Software Baseado em Frameworks de Domínio

por

VICTOR PEDROSO AMBIEL BARROS

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 05 de Novembro de 2015 como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Profª Drª Simone Nasser Matos
Orientadora

Prof. MSc. Mathias Talevi Betim
Membro titular

Prof. Dr. Ionildo José Sanches
Responsável pelos Trabalhos
de Conclusão de Curso

Prof. MSc Alessandro Luiz Stamatto
Membro titular

Prof. Dr Erikson Freitas de Morais
Coordenador do curso

- O Termo de Aprovação assinado encontra-se na Coordenação do Curso -

O único lugar onde o sucesso vem
antes do trabalho é no dicionário.
(EINSTEIN, Albert)

RESUMO

BARROS, Víctor Pedroso Ambiel. **Um Método para Refatoração de Software Baseado em Frameworks de Domínio**. 2015. 96 f. Trabalho de Conclusão de Curso Superior de Bacharelado em Ciência da Computação - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2015.

Um framework de domínio é formado por um conjunto de classes que capturam o conhecimento e especialidade em um domínio de problema particular e são criados por grupos de desenvolvedores que muitas vezes não seguem a aplicação correta dos conceitos orientados a objeto. Para melhorar a flexibilidade, legibilidade, reusabilidade, expansibilidade e manutenibilidade dos frameworks de domínio pode-se usar a refatoração de software. O processo de refatoração é facilitado quando se usa métodos específicos para a aplicação das técnicas de refatoração o que garante um melhor resultado no produto final. Métodos de refatoração na literatura ou atende uma linguagem de programação específica ou são mais abrangentes. Este trabalho criou um método de refatoração usando como referência os métodos da literatura, capaz de ajudar os desenvolvedores na refatoração de aplicações construídas com os conceitos de frameworks de domínio. O método proposto é formado por três etapas principais: Entender o sistema, Ordenar os módulos e Refatorar Módulos. A diferença entre o método proposto e os da literatura é que prevê a aplicação de metapadrões, inversão de controle e uso de ferramenta de refatoração em suas etapas. O estudo de caso em que o método foi aplicado é o Framework de Formação de Preço de Venda (FrameMK), desenvolvido pelo Grupo de Pesquisa em Sistemas de Informação do Câmpus Ponta Grossa, que tem a finalidade de calcular o preço de venda de um produto ou serviço. Os resultados da aplicação do método no FrameMK foram: melhorou a complexidade do código, diminui a quantidade de *bad smells* e a duplicação de código, o código ficou mais reusável e flexível e houve um aumento na qualidade do software em relação a expectativas do seu ciclo de vida.

Palavras-chave: Refatoração. Framework de Domínio. Método de Refatoração.

ABSTRACT

BARROS, Víctor Pedroso Ambiel. **A Method for Software Refactoring Based on Frameworks Domain**. 2015. 96 f. Trabalho de Conclusão de Curso Superior de Bacharelado em Ciência da Computação – Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2015.

A domain framework consists of a set of classes that capture the knowledge and expertise in a particular problem domain and are created by groups of developers who often do not follow the correct application of object-oriented concepts. To improve flexibility, readability, reusability, scalability and maintainability of domain frameworks can use refactoring software. The refactoring process is facilitated when using specific methods for applying the refactoring techniques which ensures a better result in the final product. Refactoring methods in the literature or answer a specific programming language or are more comprehensive. This paper created a method refactoring with reference to the methods of literature, able to assist developers in refactoring applications built with domain frameworks concepts. The proposed method consists of three main steps: Understanding the system, Sort modules and Refactor modules. The difference between the proposed method and the literature is that it provides for metapatterns, inversion of control and use of refactoring tool in their steps. The case study in which the method was applied is the Framework of Sales Price Formation (FrameMK), developed by the Research Group Information Systems on Campus Ponta Grossa, which have the purpose of calculating the selling price of a product or service. The results of applying the method in FrameMK were: improved code complexity, reduces the quantity of bad smells and the duplication code, the code became more reusable and flexible and there was an increase in the quality of software in relation to expectations of its cycle life.

Keywords: Refactoring. Domain Framework. Method Refactoring.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A refatoração de software foi apresentada por Fowler (1999) como um conjunto de técnicas que facilita modificar a estrutura interna do software, sem alterar o seu comportamento externo.

O processo de refatoração pode ser melhorado com a adoção de métodos que guiam o processo na identificação das partes do código que devem ser refatoradas e indicam qual a melhor técnica a ser utilizada em determinado tipo de problema.

Alguns métodos de refatoração já foram publicados, como o de Mens e Tourwé (2004) que possui uma sequência de seis passos a serem seguidos para se aplicar a refatoração e não é focado em um tipo de software específico. Outro trabalho é o de Rapeli (2006), que está focado na refatoração de sistemas em Java com Padrões de Projeto. Estes métodos não exploram a refatoração de aplicações construídas com o conceito de frameworks de domínio, as quais contém características que envolvem aplicação de metapadrões, padrões de projeto, inversão de controle, entre outros.

Um *framework* é uma estrutura que tem como o objetivo prover uma funcionalidade genérica, que serve de apoio para a construção de uma outra aplicação (FAYAD, 1999). Eles podem ser classificados como de infra-estrutura, *middleware* e domínio (ou aplicação). Os frameworks de infra-estrutura simplificam o desenvolvimento de sistemas de infra-estrutura portáveis e eficientes. Os frameworks *middleware* integram aplicações e componentes distribuídos, escondendo o baixo nível de comunicação entre os componentes distribuídos. Os frameworks de domínio têm um foco no desenvolvimento de aplicação em domínios específicos tais como: agricultura, formação de preço de venda, entre outros.

Este trabalho criou um método de refatoração que pode ser aplicado na estrutura de software construído com os conceitos de um *framework* de domínio, utilizando como base métodos já publicados, buscando assim obter um melhor resultado na refatoração do framework. Para isto, buscou-se quais os conceitos utilizados para a construção de um framework, assim foi possível identificar quais pontos devem ser levados em consideração em sua refatoração, introduzindo os conceitos de metapadões, inversão de controle, padrões de projeto e técnicas

de refatoração. Também foi proposto automatizar e facilitar a aplicação da refatoração utilizando ferramentas que auxiliam na análise de código.

O uso do método proposto foi aplicado no framework de domínio na Formação de Preço de Venda (FrameMK) (FRAMEMK, 2015) que está em desenvolvimento por acadêmicos da computação desta instituição. Este *framework* tem como principal objetivo oferecer um ambiente em que o usuário possa calcular o preço de venda de um produto ou serviço usando vários métodos de precificação. Foi escolhido este framework porque o código da aplicação foi desenvolvido por várias pessoas e com isto necessitava de alterações em sua estrutura interna.

Com a aplicação do método de refatoração foi possível diminuir a quantidade de *bad smells* contidos no framework, deixar o código mais reusável, reduziu-se a quantidade de códigos duplicados e a complexidade e aumentou-se sua classificação na escala *SQALE rating* (SQALE rating, 2015), que tem como objetivo medir o quão objetivo, preciso, de fácil reprodução e automatizado é o código.

1.2 OBJETIVOS

O objetivo geral deste trabalho é criar um método de refatoração para frameworks de domínio tendo como referência os métodos da literatura e etapas que tratam as características como metapadrões, inversão de controle e padrões de projeto fundamentais na arquitetura dos frameworks.

Os objetivos específicos são:

- Identificar as características de uma arquitetura de frameworks de domínio.
- Criar um guia para aplicação de metapadrões e inversão de controle em um código fonte.
- Aplicar e analisar o uso do método proposto em um estudo de caso.
- Comparar o método proposto com os da literatura.

1.3 ORGANIZAÇÃO DO TRABALHO

Este trabalho está organizado em cinco capítulos. O capítulo 2 apresenta o conceito sobre a refatoração de software, mostrando a sua importância e as técnicas existentes para a refatoração. É apresentado os dois métodos de refatoração divulgados na literatura: Mens e Tourwé (2004) e Rapeli (2006). Algumas ferramentas de análise de código são descritas neste capítulo.

O capítulo 3 aborda o conceito de *frameworks*, como pode ser feita sua modelagem e quais são as características de um sistema baseado em *framework* de domínio.

O capítulo 4 apresenta os resultados obtidos da aplicação do método no Framework de Formação de Preço de Venda (FrameMK), realiza uma análise estatística do processo de refatoração no FrameMK e compara o método proposto com os da literatura.

O capítulo 10 apresenta as conclusões deste trabalho, bem como possibilidades de trabalhos futuros relacionados ao assunto.

2 REFATORAÇÃO DE SOFTWARE

Este capítulo apresenta uma visão geral sobre refatoração de software. A seção 2.1 relata a importância e vantagens da refatoração. A seção 2.2 descreve as técnicas de refatoração publicadas na literatura. A seção 2.3 aborda o conceito de métodos que auxiliam no processo de refatoração. A seção 2.4 traz um comparativo sobre os métodos analisados. Por fim, a seção 2.5 apresenta algumas ferramentas que podem auxiliar no processo de refatoração.

2.1 A IMPORTÂNCIA DA REFATORAÇÃO

Independentemente do tipo de software que está sendo desenvolvido, ou finalizado, em um certo momento necessitará de modificação, seja para a correção de erros ou para a inclusão de novas funcionalidades. O tempo de duração de um sistema e mudanças incrementais são constantes (LEHMAN, 1980). A constante modificação de um software pode ocasionar alguns problemas, como deixar o código: desorganizado, mal codificado, com perca do desempenho, mais complexo, entre outros.

O tempo e esforço gasto para a manutenção de um software pode ser maior do que foi gasto para o seu desenvolvimento (SOMMERVILLE, 2011).

Existem técnicas que podem sem aplicadas no projeto de software que visam diminuir o esforço na manutenção e melhorar a qualidade do mesmo. Uma dessas técnicas é a refatoração de software, inicialmente apresentada por Fowler (1999), que é uma maneira de modificar a estrutura interna do software sem alterar o seu comportamento.

Refatorar um software consiste na aplicação de técnicas que estão disponíveis no catálogo de Fowler (1999), o qual é atualizado constantemente. Cada técnica abrange apenas um pequeno problema, facilitando assim a sua aplicação na reestruturação do software.

2.2 TÉCNICAS DE REFATORAÇÃO

Ao todo são 91 (noventa e uma) técnicas de refatoração que estão divididas entre 17 (dezessete) categorias, como por exemplo, existem técnicas que são de: Encapsulamento, Associações, Chamada de Métodos, Compor Métodos, entre outras. O quadro 1 apresenta apenas a categoria Compor Métodos (*Composing Methods*) que será utilizada como exemplo para explicação da técnica *Extract Method* (*Extrair Método)*. O restante das categorias com suas respectivas técnicas está no apêndice A.

Categoria	Técnica
Composing Methods	*Consoliate Conditional Expression* *Decompose Conditional* *Extract Method* *Extract Surrounding Method* *Extract Variable* *Form Template Method* *Inline Method* *Inline Temp* *Move Eval from Runtime to Parse Time* *Remove Assignments to Parameters* *Replace Loop with Collecion Closure Method* *Replace Method with Method Object* *Replace Temp with Query* *Split Temporary Variable* *Substitute Algorithm*

Quadro 1 – Categorias e técnicas de refatoração definidas por Fowler
Fonte: Fowler (1999)

A categoria *Composing Methods* possui 15 técnicas. Um exemplo de uma técnica de refatoração presente na categoria *Composing Methods* é a *Extract Methods* (*Extrair Método*) que consiste em retirar um código existente dentro de um método e criar um novo método com o código extraído (FOWLER, 1999). O código extraído é algo que não condiz com a real função do método, como por exemplo, um método para apresentar valores de produtos, onde seu valor está sendo calculado dentro do método de apresentação. Nesse exemplo, aplicando a refatoração de extração de método, a parte do código do cálculo dos valores dos produtos é extraída, gerando um novo método apenas para o cálculo e o método antigo fica correto apenas realizando a apresentação dos valores dos produtos. A figura 1 ilustra o antes e depois de se aplicar a técnica de Extrair Método.

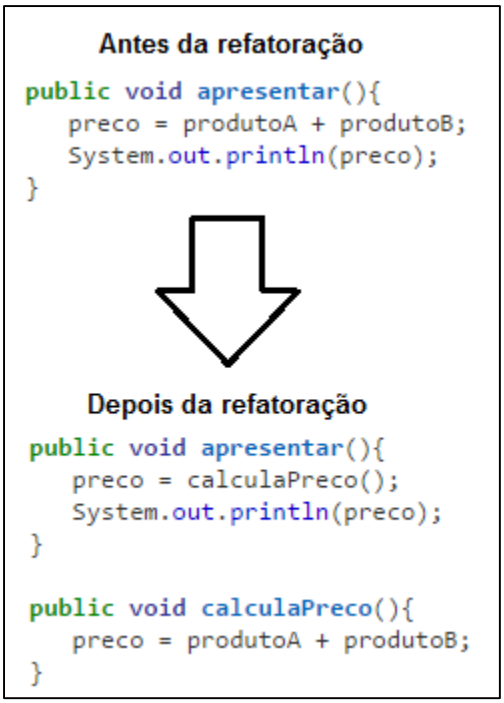


Figura 1 - Antes e depois da aplicação da técnica Extrair Método
Fonte: Autoria própria

A principal vantagem dessa refatoração é reduzir a duplicação de código, pois se vários métodos precisassem dos valores dos produtos, é necessário a implementação dos cálculos dentro de cada um dos métodos, o que gera a repetição de código.

2.3 MÉTODOS PARA REFATORAÇÃO

A refatoração pode ser aplicada de uma forma independente, identificando onde é necessário ser aplicada e qual técnica deve ser usada. Para isso o desenvolvedor deve ter conhecimento e experiência das técnicas de refatoração.

Para guiar a aplicação das técnicas de refatoração existem os métodos de refatoração que auxiliam na identificação de qual técnica deve ser utilizada em um determinado problema que ocorre no software.

Durante o estudo sobre os métodos publicados na área de refatoração identificou-se que alguns são mais específicos e outros mais genéricos. Um método genérico não se encaixa apenas a um tipo de sistema, podendo ser utilizado em diferentes linguagens e diferentes estruturas. Os métodos

específicos são voltados para um determinado tipo de sistema, com um paradigma específico, como um sistema orientado a objetos e pode ser também voltado para um tipo de linguagem de programação, tal como Java.

Dois métodos de refatoração mais citados na literatura, um específico (RAPELI, 2006) e um genérico (MENS; TOURWÉ, 2004) são detalhados nas próximas subseções.

2.3.1 Método Baseado em Padrões de Projeto de Rapeli

Rapeli (2006) apresentou sua dissertação com o foco em refatorações de sistemas construídos na linguagem Java utilizando padrões de projeto. O trabalho descreve um estudo de caso de uma refatoração de sistemas orientados a objetos, servindo como um auxílio para quem deseja aplicar a refatoração neste tipo de sistema.

Para realizar a refatoração, Rapeli (2006) propõe a utilização de um método composto por 3 (três) etapas, ilustradas na figura 2, *Entender o sistema*, *Refatorar o sistema utilizando padrões de projeto* e *Verificar sistemas após a refatoração.*

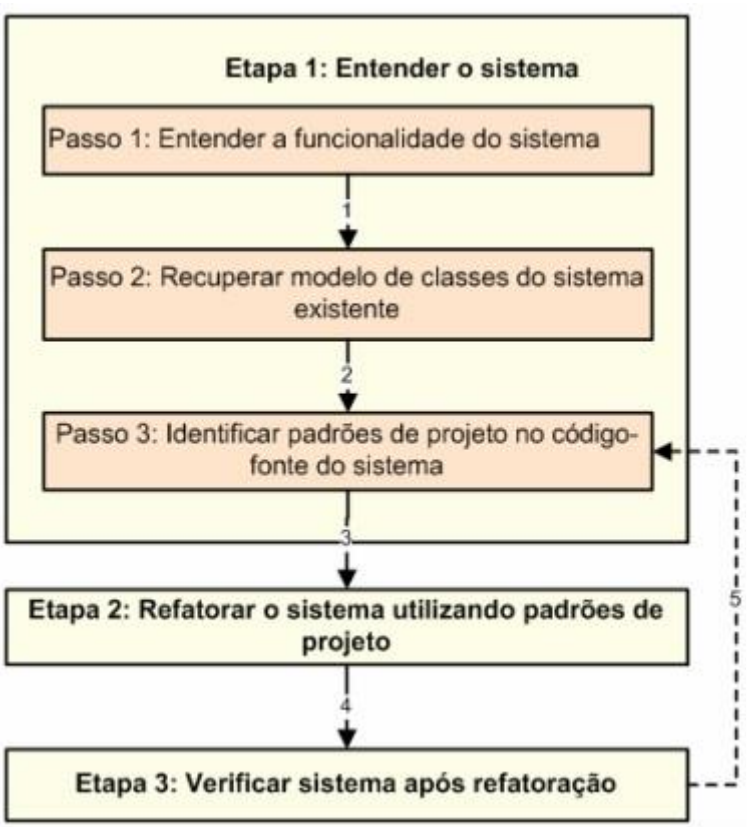


Figura 2 - Método de Rapeli Baseado em Padrões de Projeto
Fonte: Rapeli (2006, p. 26)

A primeira etapa do método tem como objetivo entender qual é a funcionalidade do sistema, para isso, existem 3 passos a serem seguidos: *Entender a funcionalidade do sistema*, *Recuperar modelo de classes do sistema existente* e *Identificar padrões de projeto no código-fonte do sistema*.

O passo de *Entender a funcionalidade do sistema* tem como objetivo compreender os requisitos funcionais do sistema, utilizando entradas e saída de dados. Para realizar esse passo, a solução é gerar uma tabela que contém:

- Número da iteração: são apenas números sequenciais para verificar quantas iterações foram realizadas
- Interação do usuário com o sistema: onde é identificado todas as interações do usuário quando o sistema é executado.
- Resposta do sistema: que representa a lista das saídas que foram fornecidas a partir da interação do usuário com o sistema.

O próximo passo é *Recuperar modelo de classes do sistema existente* que tem como finalidade recuperar o modelo de classes do sistema ou construí-lo a partir do código-fonte, sendo assim é possível reconhecer quais padrões já

estão presentes no sistema. Este passo possui duas opções, que varia de sistema para sistema. A primeira é quando o modelo de classes do sistema já existe, assim só é necessário realizar a análise para verificar se condiz com o código-fonte. A segunda opção é quando o modelo de classes do sistema ainda não está criado, assim deve-se realizar a sua construção a partir do código-fonte do sistema.

O último passo da primeira etapa é o *Identificar padrões de projeto no código-fonte do sistema,* o qual utiliza as informações contidas nos dois passos anteriores, levantando os padrões presentes no código, conforme apresentado no anexo A. Assim como o passo 2, este passo também possui duas opções. A primeira é quando o desenvolvedor já conhece em qual categoria de padrões de projeto o problema se encaixa, a partir daí deve-se procurar uma solução. A segunda opção é quando o desenvolvedor não conhece as categorias dos padrões, assim deve-se começar por qualquer categoria para identificar em qual cada uma se encaixa para obter a solução desejada.

A segunda etapa do método de Rapeli (2006) é *Refatorar o sistema utilizando padrões de projeto* que permite o refinamento do modelo de classes, sendo que a refatoração e a atualização do modelo de classes são executadas simultaneamente. Essa etapa se caracteriza por ter um conjunto de informações específicas para cada refatoração. Tal conjunto contém o nome do padrão que será utilizado, o objetivo que descreve o problema que o padrão resolverá e a solução que contém o passo a passo de como esse processo será executado.

A última etapa do método de Rapeli (2006) é *Verificar sistemas após a refatoração* que faz uma reavaliação do sistema após a refatoração. Esta reavaliação é focada no comportamento do sistema para garantir que a funcionalidade do sistema não seja alterada. Para realizar essa reavaliação, utiliza-se as iterações do passo 1 da etapa 1 para avaliar se as saídas continuam as mesmas.

2.3.2 Método de Mens e Tourwé

Mens e Tourwé (2004) mostram um método de seis passos para se aplicar a refatoração: *Identificar onde o software deve ser refatorado*, *Determinar quais*

*refatorações devem ser aplicadas nos lugares identificado*s, *Garantir que a refatoração aplicada preserve o comportamento do software*, *Aplicar a refatoração*, *Avaliar os efeitos da refatoração nas características do software* e *Manter a coerência entre o código refatorado e os outros artefatos do software*.

O passo de *Identificar onde o software deve ser refatorado,* primeiramente deve-se saber em qual nível de abstração o software será refatorado, se será aplicado no código ou em um nível mais abstrato como a sua documentação. Se a refatoração for em nível de código, tem-se duas outras atividades a serem realizadas que são: identificar quais partes do código devem ser refatorados e quais refatorações devem ser aplicadas.

O passo seguinte *Determinar quais refatorações* tem como o objetivo determinar quais as refatorações devem ser aplicadas.

A refatoração não deve modificar o comportamento do software, porém raramente se consegue manter seu comportamento. Uma maneira de se realizar os testes para garantir o comportamento é sugerida por Opdyke (1992), onde é feita a entrada de dados no software antes e depois da refatoração, e os resultados gerados devem ser idênticos. Mens e Tourwé (2004) apresentam uma forma pragmática para manter o comportamento do software usando uma disciplina de testes muito rigorosos, sendo que se o software passar por extensos conjuntos de casos de testes antes e depois da refatoração há uma grande chance de seu comportamento não ter sido modificado. Os testes são realizados no terceiro passo *Garantir que a refatoração aplicada preserve o comportamento do software*.

O passo de *Aplicar a refatoração* permite o uso das técnicas de refatoração determinadas no passo 2, nas partes do código identificados no passo 1.

Para avaliar quais os efeitos nas características do software que a refatoração causou, tais como: mudança na sua complexidade, no seu entendimento, na sua manutenibilidade, ou no seu processo de desenvolvimento como: a produtividade, o custo, entre outros, executa-se o passo 3 - *Avaliar os efeitos da refatoração nas características do software*.

O útlimo passo, *Manter a coerência entre o código refatorado e os outros artefatos do software*, é realizado para verificar se existe coerência entre o

projeto de software refatorado com os artefatos do sistema (sua documentação, arquitetura, entre outros).

2.4 ANÁLISE DOS MÉTODOS DE REFATORAÇÃO

A fase inicial do método de Rapeli (2006) foca na abstração em alto nível, sendo uma das etapas a geração do diagrama de classes do sistema que permite identificar os padrões de projeto que estão implementados ou quais ainda podem ser usados. Por outro lado, o método de Mens e Tourwé (2004) não é focado apenas em uma abstração em alto nível, onde fica a critério de quem irá utilizar o método, definir qual será o nível de refatoração, sendo assim um método mais flexível.

Na aplicação da refatoração pode haver diferença entre os dois métodos, que depende do nível de abstração que for escolhido. No método de Mens e Tourwé (2004) é possível realizar a refatoração sem que haja uma grande modificação na estrutura de classes do sistema, quando realizado em baixo nível de abstração, já no método de Rapeli (2006) a alteração na estrutura de classes é determinada com base na análise que é realizada no diagrama de classes. O quadro 2 apresenta a comparação entre os dois métodos.

Características	Rapeli (2006)	Mens e Tourwé (2004)
Compreender a funcionalidade do sistema.	X	X
Gerar diagrama de classe.	X	
Aplicar padrões de projeto de forma específica.	X	
Testar sistema refatorado.	X	X
Refatorar para qualquer linguagem.		X
Analisar código para refatorar.	X	X
Identificar tipo de refatoração a serem aplicadas.		X

Quadro 2 - Comparação entre o método de Rapeli (2006) e Mens e Tourwé (2004)
Fonte: Autoria própria

No quadro 2 observa-se com maior facilidade que existem características que o método de Rapeli (2006) não aborda, como por exemplo a identificação dos tipos de refatorações que podem ser aplicadas no sistema. E existem características que o método de Mens e Tourwé (2004) também não

contempla como a aplicação de padrões de projeto de forma específica, contemplado no método de como Rapeli (2006). Nota-se também que nenhum método tem a preocupação em analisar características específicas de aplicações baseadas em framework de domínio, descritas detalhadamente na seção 3.3.

2.5 FERRAMENTAS PARA A REFATORAÇÃO

Existe várias ferramentas disponíveis para auxiliar no processo de refatoração, neste trabalho serão listadas e abordadas 4 (quatro) ferramentas, sendo todas gratuitas, pois o Grupo de Pesquisa em Sistemas de Informação (GPSI), para o qual este trabalho será usado, tem como meta o desenvolvimento de produtos usando ferramentas que não pagas. Algumas das ferramentas para a análise do código são: *SonarQube* (2008), *Checkstyle* (2001), *FindBugs* (2003) e *CodePro AnalytiX* (2001).

SonarQube (2008) é uma ferramenta de código aberto que realiza uma análise completa de um projeto ou sistema, possibilitando um controle sobre um grande número de métricas de software, apontando também *bugs* do sistema, mostrando partes onde o código está duplicado e poderia ser melhorado, tudo sendo exibido através de uma interface web, onde os resultados são mostrados graficamente. A ferramenta está disponível para mais de 25 linguagens de programação, como Java, C, C++, PHP, C#, entre outras, além de ser usada para o desenvolvimento *mobile*, especificamente para a plataforma *Android*.

Checkstyle (2001) é uma ferramenta que auxilia na codificação em Java. Ela analisa problemas no *layout* do código, busca partes de códigos que estão duplicados, faz uma busca por *bugs*, entre outras possibilidades. Para se utilizar a ferramenta, deve-se informar quais os módulos desejados para realizar a análise. Os problemas que forem encontrados são mostrados na aba “Problemas” do Eclipse, onde também é exibido uma sugestão para resolvê-lo. A ferramenta possui um módulo de métricas e permite construir um módulo customizado de análise de código.

FindBugs (2003) é uma ferramenta de código aberto mantida pela *University of Mariland*. Ela possui recursos como: a análise do código, busca por *bugs*, entre outros, mas o seu diferencial é que classifica os problemas em

diversas categorias, permitindo identificar quais tipos de *bugs* devem ser buscados e isto possibilita focar nos principais problemas a serem corrigidos. A ferramenta apresenta o problema que pode acontecer no trecho do código examinado, mostrando também individualmente uma descrição sobre o *bug* encontrado e o porquê é ruim para o código. Ela está disponível somente para a linguagem Java e é utilizada em projetos grandes e mais complexos.

CodePro AnalytiX (2001) é uma ferramenta que apresenta recursos similares a *FingBugs*. É mantida pelo *Google* em uma parceria com o *Eclipse Project*. A similaridade dela com as ferramentas anteriores é que mostra o problema, o porquê deste ser ruim para o código e faz uma sugestão de melhoria. Ela também possui um módulo de geração de casos de testes automatizados e no seu módulo de métricas permite a visualização de gráficos e a configuração de valores considerados limites para cada métrica. Quando esse limite é ultrapassado, o relatório da métrica é apresentada em vermelho para evidenciar o problema encontrado.

Várias ferramentas podem ajudar no processo de refatoração, porém deve-se escolher aquela que se adapta melhor ao tipo do projeto ou sistema.

O primeiro critério de comparação é se a ferramenta pode ser utilizada em diferentes linguagens de programação, como Java, C#, C++, entre outras. O segundo critério de comparação é se a ferramenta apresenta uma sugestão de solução para os problemas que foram encontrados no código. O terceiro critério de avaliação é se a ferramenta é independente de uma IDE, como Eclipse ou *NetBeans*, tanto para analisar o código quando para apresentar os resultados da análise.

Dentre as ferramentas apresentadas anteriormente, todas foram testadas em uma pesquisa de iniciação científica (BARROS, 2014) e foi selecionada primeiramente a *CodePro Analyix*, porém, posteriormente foi identificado que a ferramenta *SonarQube* consegue detectar a maior quantidade de *bad smells*. Portanto, esta será a ferramenta utilizada neste trabalho.

É importante ressaltar que o método proposto não estabelece qual ferramenta de refatoração que o desenvolvedor deve usar, porém indica que uma deve ser adotada durante o processo.

3 FRAMEWORK DE DOMÍNIO

Este capítulo aborda conceitos sobre *framework* de domínio. A seção 3.1 descreve os conceitos sobre *framework*, benefícios de sua utilização e classificação. A seção 3.2 aborda como pode ser feita a modelagem de *frameworks* de domínio. A seção 3.3 apresenta as características de um sistema baseado em *framework* de domínio. A seção 3.4 apresenta informações sobre o framework de domínio que será refatorado usando o método proposto.

3.1 CONCEITOS E BENEFÍCIOS DE FRAMEWORKS

Em geral, um *framework* é uma estrutura que tem como o objetivo prover uma funcionalidade genérica, que serve de apoio para a construção de uma outra aplicação (FAYAD; SCHMIDT, 1997).

Os *frameworks* frequentemente são definidos como um projeto reutilizável ou parte de um sistema que é representado por um conjunto de classes abstratas e a forma como as suas instâncias interagem (FAYAD *et al*., 1999).

Diferentemente de uma biblioteca, que pode ser definida como um conjunto de implementações, seja de classes, funções, procedimento, entre outros, o *framework* engloba outras tarefas tal como: estabelece o fluxo de controle da aplicação (Inversão de Controle) (FOWLER, 1999). Um *framework* também pode ser construído a partir da união de várias bibliotecas.

Dentre os benefícios da utilização de um *framework*, estão listados os quatro mais relevantes (FAYAD; SCHMIDT, 1997):

- *Modularidade*: encapsula detalhes de implementação por meio dos pontos de extensão e das interfaces estáveis e bem definidas, tem-se um aumento da modularidade. Com os locais de mudanças de projeto e a implementação da aplicação construída sendo especificados, diminui-se o esforço para entender e manter a aplicação, que por sua vez melhora a qualidade do software.
- *Reusabilidade*: fornece interfaces estáveis, pois permitem definir componentes genéricos para a criação de novas aplicações. Com

a maturidade dos componentes reusáveis, incrementa-se três outros benefícios: qualidade, desempenho e confiança no funcionamento.

- *Extensabilidade*: os *frameworks* fornecem pontos de extensão que possibilitam aos desenvolvedores estenderem as funcionalidades para poder gerar uma nova aplicação. Fornecer os pontos de extensão é essencial para garantir a personalização de novos serviços e funcionalidade da aplicação.
- *Inversão de controle*: não é uma característica de todos os *frameworks*, porém alguns apresentam esse benefício. A inversão de controle (*Inversion of Controll*, IoC) transfere o controle de execução da aplicação para o *framework*, controlando em decorrência de algum evento quais métodos da aplicação serão chamados e em que contexto.

Os *frameworks* são compostos pelos pontos de estabilidade e flexibilidade, também chamados de pontos fixos e pontos extensíveis, ou *frozen spots* e *hot spots*. Os pontos de estabilidade são as funcionalidades e serviços que já estão implementados pelo *framework*, já os pontos de flexibilidade são as funcionalidades e serviços que devem ser implementados por quem irá utilizar o *framework*, utilizando a técnica de herança para inserir códigos que serão inerentes ao domínio de aplicação do *framework* (FAYAD; SCHMIDT, 1997).

Os *frameworks* podem ser classificados em três tipos: infra-estrutura, integração *middleware* e de aplicação (FAYAD; SCHMIDT, 1997). Os *frameworks* de infra-estrutura simplificam o desenvolvimento de sistemas de infra-estrutura portáveis e eficientes, como interfaces gráficas e sistemas operacionais. São conhecidos como frameworks horizontais porque não se referem apenas a um domínio de aplicação específico, sendo assim são mais genéricos, podendo ser aplicados a várias áreas, como por exemplo frameworks para a construção de interfaces gráficas, realizar persistência, entre outras (FAYAD; SCHMIDT, 1997).

Os *frameworks* de integração *middleware* integram aplicações e componentes distribuídos, escondendo o baixo nível de comunicação entre os componentes distribuídos, o que permite aos desenvolvedores em um ambiente distribuído seja semelhante a um não distribuído.

Os *frameworks* de aplicação têm um foco diferente dos dois outros apresentados, sendo ele voltado para a aplicação em domínios específicos, por exemplo um domínio na área financeira. Estes são também conhecidos como frameworks verticais, também chamados de especialistas, pois resolvem apenas o problema de uma área.

3.2 MODELAGEM BASEADA EM FRAMEWORK DE DOMÍNIO

Na literatura analisada se identificam abordagens de processos de desenvolvimento de *frameworks* de domínio, sendo os principais definidos por Matos e Fernandes (2007): Johnson (1993), Wilson e Wilson (1993), Taligent (1994), Landin e Niklasson (1995), Mattson (1996), Pree (1999), Fayad et al. (1999), Silva (2000), Butler *et al.* (2002), Braga (2003), Ben-Abdallah *et al*. (2004) e Camargo e Masiero (2005).

Johnson (1993), Wilson e Wilson (1993) propõem que por meio de aplicações-exemplo ou análise de aplicações já testadas se busque as funcionalidades que são iguais e as agrupam em classes.

Analisar o domínio de pequenos *frameworks* é proposto por Taligent (1994), abordando o desenvolvimento de *frameworks* mais simples, construindo aplicações-exemplo a partir do domínio analisado.

Landin e Niklasson (1995) propõem realizar a análise dos requisitos, classes e objetos que são comuns nas aplicações-exemplo.

Mattson (1996) propõe seguir uma sequência de fases baseando-se na análise do projeto e tendo como base a arquitetura do *framework*. Para chegar nos objetivos de extensibilidade, generalidade e alterabilidade é feita a construção de novos projetos utilizando as bases citadas, através de iterações que repetem as fases até que se chegue em uma estrutura de classes que atenda aos requisitos.

Pree (1999) aborda que para que um projeto atinja um ponto de estabilidade ele deve atender os requisitos dos pontos de flexibilidade, para isso evidência que os pontos de flexibilidade na estrutura de classe é o requisito principal.

Fayad *et al.* (1999) propõe uma abordagem baseada em Desenvolver, Usar e Evoluir. Na fase *Desenvolver*, é realizado o desenvolvimento da arquitetura do projeto, o projeto do *framework*, a implementação, os testes e a documentação e por fim é realizado a utilização. Na fase *Usar*, utiliza-se as aplicações-exemplos baseado na arquitetura de aplicação, análise de requisitos e a composição de componentes. Na fase *Evoluir* é realizada a manutenção do *framework*, permitindo alterações que não foram previstas no início do desenvolvimento, como mudanças no negócio, na aplicação ou no domínio.

A abordagem de Silva (2000) é a mesma realizada por Mattson (1996), porém as fases não são sequenciais.

O processo proposto por Butler *et al*. (2002) é baseado em refatoração e possui várias fases de evolução aplicadas aos aspectos de domínio do *framework*. Primeiramente são identificados os pontos de estabilidade e flexibilidade, e o diagrama de caso de uso. Posteriormente são aplicados no código os métodos-gabarito e métodos-gancho obtendo uma maior flexibilidade no projeto. Os métodos-gabarito e métodos-gancho serão explicados mais detalhadamente na seção 3.3.

A abordagem de Braga e Masiero (2003) propõe a aplicação de linguagens padrões, que são linguagens que tem como foco a construção de *frameworks*, tendo como função principal desenvolver uma escrita que possa criar uma estrutura de software ou sistemas de informação.

O processo proposto por Ben-Abdallah *et al.* (2004), baseado em UML (*Unified Modeling Language*), propõe o desenvolvimento do *framework* por meio de diagramas de classe, de caso de uso e de sequência, porém não há nenhum fluxo que os interliguem, sendo assim de forma independente.

Camargo e Masiero (2005), propõem uma abordagem baseada em aspectos, definindo que um *framework* orientado a aspectos "é um conjunto formado geralmente por classes e aspectos". Na abordagem de Camargo e Masiero (2005) também evidencia os pontos de estabilidade e flexibilidade.

O estudo das abordagens relatadas anteriormente foi importante para o levantamento das características que um sistema baseado em frameworks deve possuir. Essas características estão explicadas na próxima subseção e são base para a criação do método proposto neste trabalho.

3.3 CARACTERÍSTICAS DE SISTEMAS BASEADOS EM FRAMEWORK DE DOMÍNIO

Durante os estudos sobre framework identificou-se que suas principais características é possuir: padrões de projeto, metapadrões e inversão de controle. A inversão de controle tem como objetivo transferir o controle das instâncias de uma classe para uma outra classe, interface ou componente, sendo esses externos. Neste caso, a instância da classe será tratada externamente e não dentro da própria classe que irá utilizá-la.

Padrão de projeto é o encapsulamento da descrição abstrata e estruturada de uma solução satisfatória para um problema que ocorre repetidamente dentro de um contexto, dado um conjunto de forças ou restrições que atuam sobre ele (GAMMA *et al*., 1994).

Os padrões de projeto têm como benefício a diminuição da complexidade do software que garante maior reusabilidade e aumento da produtividade dos desenvolvedores. Em relação ao desenvolvimento de *frameworks* de domínio, que possuem uma alta complexidade (SILVA, 2000), sendo assim, auxiliam na flexibilidade do software.

Não somente os padrões de projeto podem auxiliar na construção de um *framework* de domínio, mas também os metapadrões, que foram apresentados por Pree (1995) como um conjunto de padrões de projeto que descreve como construir *frameworks*, independente de um domínio de aplicação específico. Os metapadrões não substituem os padrões de projeto, mas são um complemento.

O padrão de projeto *Template Method* (GAMMA *et al*. 1994) é o responsável pela origem dos metapadrões. Para exemplificar o *Template Method*, tem-se o problema da reprodução de uma lista de músicas, onde essa lista pode ser reproduzida ordenada por nome da música, nome do autor ou por ano da música (BRIZENO, 2011). Para resolver esse problema, outros padrões de projeto poderiam ser utilizados, como o *Strategy*, definindo um método de reprodução para cada tipo (nome da música, do autor e ano), porém, o algoritmo que irá realizar a reprodução da música é o mesmo nos três casos, a única diferença entre eles é que cada um leva em consideração um dos atributos da música. A solução é utilizar o *Template Method*, em que o *método-gabarito* implementa o algoritmo de reprodução das músicas e deixaria o *método-gancho*

decidir qual seria a ordem de execução, sendo este um exemplo do metapadrão *1:1 Unification*.

O *método-gabarito* tem como objetivo definir o esqueleto dentro de um método, transferindo algumas de suas funcionalidades para as suas subclasses o que permite que se redefinam certos passos de um algoritmo sem que seja necessário alterar a estrutura do mesmo (GAMMA *et al*., 1994). O *método-gancho* serve como uma ligação entre as subclasses e o algoritmo. Ele é declarado na classe-gabarito e é por padrão uma implementação vazia, permitindo que as subclasses se conectem ao algoritmo em vários pontos.

Os métodos-gabaritos implementam os pontos comuns do sistema e os métodos-gancho são responsáveis pelos pontos variáveis. Existem vários casos em que é melhor colocar em classes separadas os pontos comuns e variáveis. Com a separação, tem-se o conceito de classes-gabarito e classes-gancho. Uma classe-gancho é aquela que contém o *método-gancho* sendo que esse método possui o *método-gabarito* correspondente. A classe-gabarito é a que contém o *método-gabarito* que irá utilizar os serviços de uma *classe-gancho* (PREE; 1995).

3.4 FRAMEWORK DE FORMAÇÃO DE PREÇO DE VENDA (FrameMK)

O método proposto por este trabalho será aplicado no FrameMK, *framework* de domínio, que possui como objetivo estabelecer preço de venda de um produto ou serviço (MAZER JUNIOR, 2013). Para calcular o preço de venda do produto ou serviço, o FrameMK possui vários métodos implementados, sendo eles o método Sebrae, o Custo Pleno e o ABC (ANDRADE; CAPELLER; MATOS, 2010).

O *framework* vem sendo desenvolvido desde 2008 pelos acadêmicos da Universidade Tecnológica Federal do Paraná câmpus Ponta Grossa que integram o GPSI (Grupo de Pesquisa em Sistemas de Informação), sendo assim, há uma grande rotatividade de pessoas que já trabalharam e ainda realizam pesquisas no *framework*. Devido essa rotatividade de pessoas, surgiu a necessidade de realizar a refatoração em sua estrutura.

O desenvolvimento do FrameMK é realizado na linguagem Java, utilizando o *framework* de aplicação *Java Struts* para Web e Java Swing para a versão desktop (MAZER JUNIOR, 2013). A página inicial do FrameMK em funcionamento é apresentada na figura 3 em que se exibe sua finalidade.

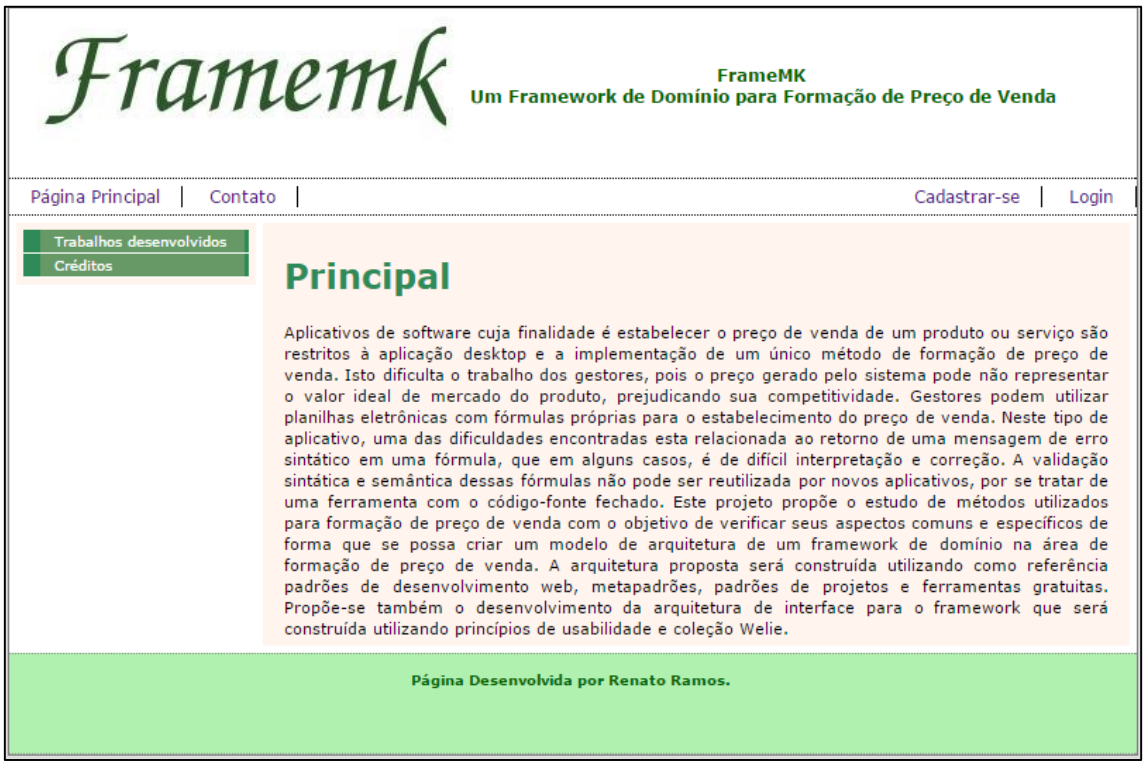


Figura 3 - Página de início do FrameMK
Fonte: Autoria própria

O FrameMK possui uma arquitetura, Figura 4, que possibilita que empresas acessem o servidor do FrameMK a partir de uma conexão de internet, sendo assim possível utilizar as funções do *framework* por intermédio de uma camada de serviços (MAZER JUNIOR, 2013).

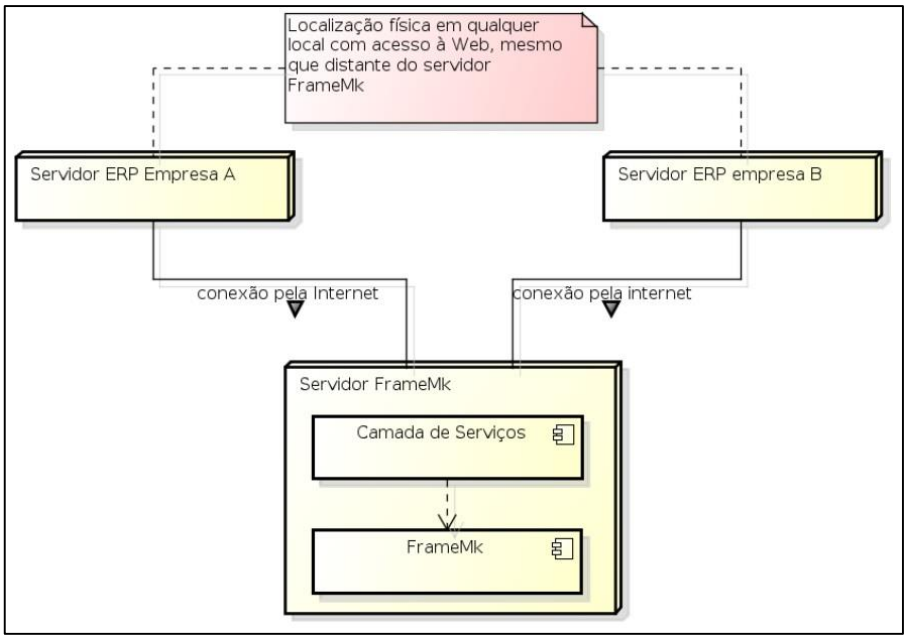


Figura 4 - Arquitetura do FrameMK
Fonte: Mazer Junior (2013, p. 76)

A figura 4 ilustra o como empresas podem utilizar o FrameMK a partir de uma conexão com a Internet. Primeiramente o servidor de uma determinada empresa se comunica com o servidor do FrameMK, posteriormente a camada de serviços faz a ligação entre o servidor da empresa e as funções do FrameMK, possibilitando assim que a empresa utilize o *framework*.

A estrutura do FrameMK é composta por cinco conjuntos de pacotes principais que exercem funções essenciais para o seu funcionamento (RIBAS, 2014).

O primeiro conjunto de pacotes é o *app*, que é a parte de apresentação do *framework*, sendo responsável pela sua parte visual.

O segundo conjunto é composto pelo pacote *BussinesRule*, onde se encontra as principais classes de código como as regras e lógicas, configurações e cálculos dos métodos de custeio e os pacotes *Persistence*, que é composto pela lógica que realiza a persistência das informações no banco de dados.

O terceiro conjunto de pacote é o *web services* que contém a implementação do serviço web do FrameMK. O terceiro pacote é o *test*, local em que se tem as classes de teste do sistema web.

O quinto conjunto de pacotes do FrameMK é o *sistemasPOA*, onde é implementado o sistema de *login*, sendo esse sistema codificado em orientação a aspectos.

4 MÉTODO PARA REFATORAÇÃO DE FRAMEWORKS

Este capítulo apresenta o método proposto para a refatoração de frameworks de domínio, utilizando como base os métodos apresentados na seção 2.3.1 e 2.3.2. A seção 4.1 apresenta uma visão geral do método proposto. A seção 4.2 descreve a primeira etapa do método *Entender o sistema*. A seção 4.3 relata o funcionamento da etapa de *Ordenar módulo*. A seção 4.4 narra a etapa em que se realiza a refatoração efetivamente no módulo.

4.1 PROCESSO GERAL DO MÉTODO PROPOSTO

O método proposto (Figura 5) é dividido em três etapas principais, sendo que para cada uma foi criado um fluxograma explicitando o seu funcionamento.

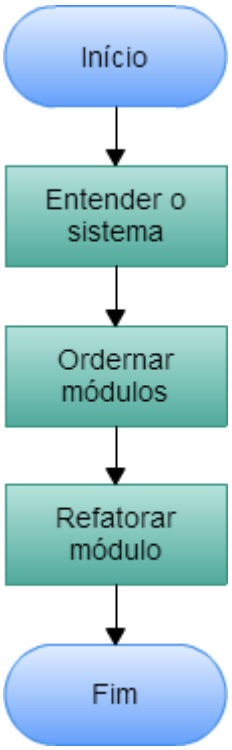


Figura 5 – Processo Geral do Método Proposto
Fonte: Autoria própria

A primeira etapa *Entender o sistema* tem a finalidade de permitir ao desenvolvedor um entendimento geral de como o sistema funciona. A segunda etapa, *Ordenar módulos*, tem como objetivo classificar os módulos que

compõem o sistema para que estes possam ser refatorados na etapa posterior. Por fim, a última etapa *Refatorar módulo* realiza o processo refatoração propriamente dito.

Dentre as etapas do método, a primeira está baseada no método de Rapeli (2006), que contém a etapa de *Entender o sistema* e o passo de *Gerar diagramas de classe*. As outras etapas foram determinadas por este trabalho.

4.2 ETAPA 1 – ENTENDER O SISTEMA

A primeira etapa *Entender o sistema* é composta por um conjunto de passos ilustrados pelo fluxograma da figura 6.

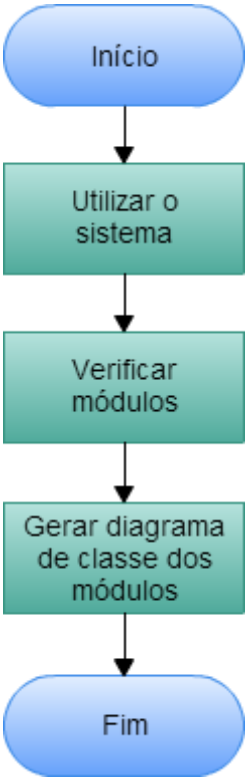


Figura 6 - Detalhamento da etapa de Entender o Sistema
Fonte: Autoria própria

Esta etapa consiste em três passos que são essenciais para a realização da refatoração do sistema. O primeiro passo é o *Utilizar o sistema*, onde seu objetivo é que o desenvolvedor tenha o primeiro contato com o sistema que será refatorado, verificando como ele funciona e interage.

O segundo passo é o *Verificar módulos*, tem como objetivo identificar quais são os módulos do sistema. No método proposto foi considerado um módulo como sendo um subsistema ou pacote. Muitos sistemas são divididos em subsistemas, como por exemplo, um *Enterprise Resource Planning* (ERP), que possui módulos que se interagem entre si, tais como: Fiscal, Financeiro, entre outros. Separar o sistema em módulos facilita o entendimento sobre seu funcionamento. Muitas vezes a separação por módulos não é explícita, sendo assim, pode-se considerar um módulo um conjunto de pacotes que tem como objetivo fornecer uma funcionalidade específica, como por exemplo, um pacote de Regra de Negócio que contempla todas as classes de regras lógicas do sistema.

O terceiro passo é o *Gerar diagrama de classe dos módulos*. O objetivo deste passo é criar o diagrama de classe dos módulos que foram identificados no passo anterior, podendo assim, ter uma representação visual e mais clara de como as classes do sistema funcionam e se relacionam entre si.

4.3 ETAPA 2 – ORDENAR MÓDULO

A segunda etapa de alto nível do processo de refatoração é *Ordenar módulo*, nesta é escolhido o módulo que será refatorado e se baseia no algoritmo de ordenação *Insertion Sort*. Para isso, tem-se o fluxograma da Figura 7 que apresenta como a escolha do módulo pode ser realizada.

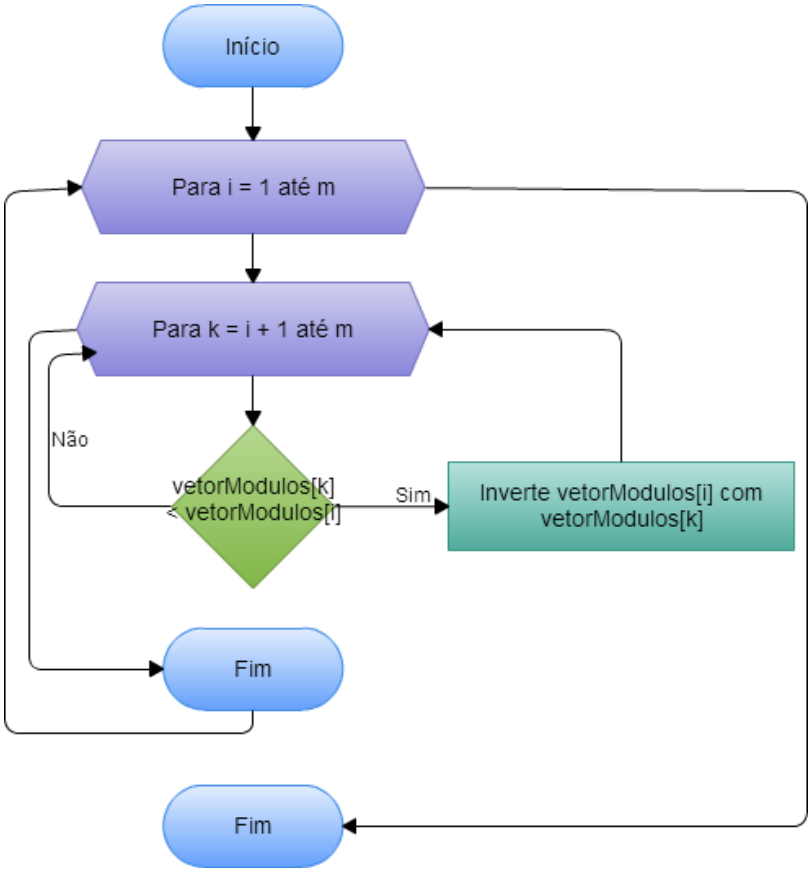


Figura 7 - Escolha do módulo que será refatorado
Fonte: Autoria própria

Esta etapa do processo de refatoração tem como foco ordenar de forma crescente os módulos pela quantidade de *bad smells*.

Para se obter a quantidade de *bad smells* de cada módulo é utilizada uma ferramenta de análise tais como: *SonarQube (2008)*, *FindBugs (2003)*, *Checkstyle (2001)* e *CodePro AnalytiX (2001)*, descritas na seção 2.5.

O fluxograma inicia com a uma estrutura de repetição *i* até *m*, onde *m* representa a quantidade total de módulos, sendo assim, todos os módulos serão verificados. O segundo *loop*: *k = i + 1* até *m* vai do índice em que *i* está até *m*. Estas duas estruturas formam um algoritmo simples de ordenação. A próxima condição expressa por **(1):**

$$vertorModulos[k] < vetorModulos[i] \quad \textbf{(1)}$$

tem como finalidade verificar se a quantidade de *bad smells* do módulo atual *k* é menor que a quantidade de *bad smells* do módulo *i*. Se for, tem-se a inversão entre os valores do *vetorModulos[i]* com *vetorModulos[k]* **(2)**:

Inverte *vetorModulos[i]* com *vetorModulos[k]* **(2)**

que inverte a posição do módulo *k* com *i*. Essa inversão faz com que o vetor de módulo permaneça em ordem crescente de quantidade de *bad smells*. Caso a verificação seja falsa, a iteração continua sem realizar nenhuma alteração.

Ao final de todas as iterações, quando todos os módulos já estiverem avaliados, tem-se o vetor de módulo ordenado do menor para o maior, considerando a quantidade de *bad smells*.

A escolha de começar pelo módulo que contém a menor quantidade de *bad smells* pode facilitar a refatoração para desenvolvedores menos experientes, pois o número de refatorações que deverão ser feitas tendem a ser menores.

4.4 ETAPA 3 – REFATORAR MÓDULO

A última etapa do método proposto é *Refatorar Módulo e* é o local de fato onde o código é refatorado. O processo desta etapa está exibido na figura 8.

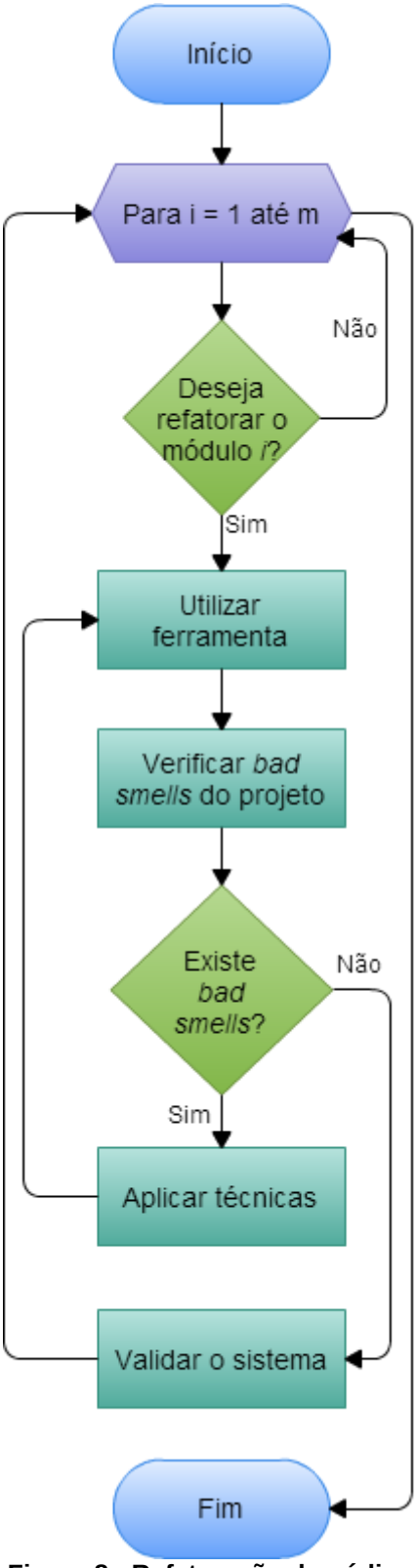


Figura 8 - Refatoração do código
Fonte: Autoria própria

Esta etapa inicia com a iteração de *i* começando em 1 e indo até *m*. Dentro da iteração verifica-se se o desenvolvedor deseja refatorar o módulo *i*, onde os

módulos estão ordenados em ordem crescente da quantidade de *bad smells*, conforme a etapa 2.

Se o desenvolvedor desejar refatorar o módulo, então é feita a utilização da ferramenta para analisar o projeto (módulo escolhido na etapa anterior), porém agora o foco é refatorar o módulo propriamente.

Após a análise do projeto a partir da ferramenta, é verificado os *bad smells* existentes, e tem-se uma condição de verificação. Se há *bad smells* no sistema, o próximo passo é aplicar as técnicas de refatoração, essas focadas em frameworks. Tais técnicas são: as técnicas de refatoração apresentadas por Fowler (1999) (conforme apresentadas na seção 2.2), aplicação de padrões de projeto (anexo A), aplicação de metapadrões e aplicação de inversão de controle (IC). A escolha de qual técnica deve ser realizada inicialmente fica a critério do desenvolvedor a medida que ele analisa o código e consegue identificar a aplicação de alguma técnica.

Como as técnicas de refatoração de Fowler (1999) e aplicações de padrões já foram publicadas na literatura, explica-se como será o processo de refatoração usando metapadrões e IoC.

A refatoração com metapadrões é apresentada nos quadros 3-9 e a inversão de controle no quadro 10.

<table>
<tr><th>Metapadrão</th><th>Unification</th></tr>
<tr><td>Descrição</td><td>No metapadrão Unification, a classe-gabarito e a classe-gancho estão unificadas em uma única classe, o que faz com que o desenvolvedor sobrescreva o método-gancho (MATOS, 2008).</td></tr>
<tr><td>Problema Endereçado</td><td>O metapadrão Unification é definido quando há duas ou mais subclasses que estendem a mesma superclasse e estas subclasses implementam métodos parecidos, sendo a diferença entre esses métodos apenas a forma de implementação, que depende da finalidade da sua classe (GURA; CARMO, 2009). O exemplo de código utilizado é quando as classes implementam o mesmo método sendo que a diferença entre as classes é o domínio de aplicação e o comportamento do método. Um exemplo de código é apresentado abaixo. Neste exemplo tem-se duas classes que implementam um método idêntico (Método 2), e outro método parecido (Método 1) sendo a diferença é que no método da Classe1 é implementado um algoritmo denominado Implementação 1 e na Classe2 é implementado o algoritmo Implementação 2.
<pre>
public class Classe1 {
    ... //Outras implementações
    public void metodo1(){
        //Implementação 1
    }

    public void metodo2(){
        //Implementação 3
    }
    ... //Outras implementações
}
</pre>
<pre>
public class Classe2 {
    ... //Outras implementações
    public void metodo1(){
        //Implementação 2
    }

    public void metodo2(){
        //Implementação 3
    }
    ... //Outras implementações
}
</pre>
</td></tr>
<tr><td>Solução</td><td>A solução é que na superclasse seja definida a assinatura do método parecido e as subclasses que estendem essa superclasse sobrescrevem esse método (Método 1), e que na superclasse seja definida a implementação do método idêntica (Método2) e as subclasses utilizem esse método, como apresentado no código abaixo.</td></tr>
</table>

```
public abstract class SuperClasse {
    ... //Outras implementações
    public abstract void metodo1();

    public void metodo2(){
        //Implementaçção 3
    }
    ... //Outras implementações
}
```

```
public class SubClasse1 extends SuperClasse {
    ... //Outras  implementações
    @Override
    public void metodo1(){
       //Implementação 1
    }
    ... //Outras implementações
}
```

```
public class SubClasse2 extends SuperClasse {
    ... //Outras  implementações
    @Override
    public void metodo1(){
       //Implementação 2
    }
    ... //Outras implementações
}
```

No exemplo de implementação, tem-se a classe *SuperClasse* que contém o método *metodo1* e *metodo2* e duas subclasses *SubClasse1* (*Classe1*) e *SubClasse2* (*Classe2*) implementam o *metodo1* definido na superclasse, onde cada subclasse implementa o método conforme a sua finalidade, e a agora as subclasses somente utilizam o *metodo2* que é implementado na *SuperClasse*. Como o intuito deste metapadrão é unificar as *classes-gabarito* e gancho, no exemplo de solução essas duas classes são representadas pela superclasse.

Quadro 3 – Refatoração Baseada no Metapadrão Unification
Fonte: Autoria própria

<table>
<tr><th>Metapadrão</th><th>1:1 Connection</th></tr>
<tr><td>Descrição</td><td>O metapadrão 1:1 Connection é definido quando há uma ligação entre duas classes, onde a primeira classe (gabarito) refere-se a exatamente uma instância de segunda classe (gancho), e não há um relacionamento de herança entre essas classes. Este metapadrão permite que o comportamento dos objetos possam ser modificados em tempo de execução (MATOS 2008).</td></tr>
<tr><td>Problema Endereçado</td><td>O metapadrão 1:1 Connection deve ser utilizado se deseja obter um relacionamento um para um entre duas classes, ou seja, quando se deseja que subclasses possam implementar partes de algoritmos que sejam pertinentes a sua aplicação e com isto se evita duplicação de código, agrupando as subclasses em uma classe única (GURA; CARMO, 2009). No exemplo apresentado no código abaixo tem-se as classes Classe2 e Classe3 que pertencem à um mesmo domínio de aplicação, portanto, implementam métodos parecidos, porém cada uma conforme a sua necessidade, não há nenhuma relação entre elas e as duas classes mantem um relacionamento 1:1 com a Classe1.
<table>
<tr>
<td><pre>public class Classe1 {
    ... //Outras implementações
    Classe2 classe2 = new Classe2();
    Classe3 classe3 = new Classe3();
    ... //Outras implementações
}</pre></td>
<td><pre>public class Classe2 {
    ... //Outras implementações
    public void metodo1(){
        //Implementação do método 1
    }
    ... //Outras implementações
}</pre></td>
<td><pre>public class Classe3 {
    ... //Outras implementações
    public void metodo1(){
        //Implementação do método 1
    }
    ... //Outras implementações
}</pre></td>
</tr>
</table>
A solução para o código apresentado utilizando o metapadrão 1:1 Connection é transformar as classes Classe2 e Classe3 em subclasses, definindo os métodos em uma superclasse, e cada subclasse implementaria o método da superclasse conforme a sua necessidade.</td></tr>
<tr><td></td><td>Para solucionar o problema apresentado, foi criada a classe-gabarito Classe (antiga Classe1) e as classes Classe2 e Classe3 agora foram transformadas em SubClasse1 e SubClasse2, como ilustrado no código abaixo.</td></tr>
</table>

<table>
<tr><td>Solução</td><td>
<table>
<tr>
<td>

```
public class Classe {
   ... //Outras implementações
   SuperClasse superclasse = new SuperClasse();

   public void metodo1(){
      //Implementação do método 1
   }
   ... //Outras implementações
}
```

</td>
<td>

```
public abstract class SuperClasse extends Classe {
   ... //Outras implementações
   public abstract void metodo1();
   ... //Outras implementações
}
```

</td>
</tr>
<tr>
<td>

```
public class SubClasse1 extends SuperClasse {
   ... //Outras implementações
   public void metodo1(){
      //Implementação do método 1
   }
   ... //Outras implementações
}
```

</td>
<td>

```
public class SubClasse2 extends SuperClasse {
   ... //Outras implementações
   public void metodo1(){
      //Implementação do método 1
   }
   ... //Outras implementações
}
```

</td>
</tr>
</table>

Foi definida também uma classe abstrata como gabarito denominada como SuperClasse, onde é feita a definição de cada método que as subclasses poderão implementar. Entre a Classe (gabarito) e a SuperClasse (gancho) há uma relação um para um (1:1), onde dentro da classe-gabarito há uma referência de objeto da classe-gancho. Com a classe-gabarito e gancho definidas, são definidas as subclasses que irão implementar os métodos definidos na classe-gancho, onde cada subclasse irá implementar o método conforme a finalidade da sua aplicação.
</td></tr>
</table>

Quadro 4 – Refatoração Baseada no Metapadrão 1:1 Connection
Fonte: Autoria própria.

<table>
<tr><th>Metapadrão</th><th>1:N Connection</th></tr>
<tr><td>Descrição</td><td>O metapadrão 1:N Connection se assemelha bastante com o 1:1 Connection, sendo a única diferença entre eles é que não somente um objeto pode ser instanciado, e sim uma lista de objetos, tendo assim uma relação 1 para N (MATOS 2008).</td></tr>
<tr><td>Problema Endereçado</td><td>O metapadrão 1:N Connection deve ser utilizado se deseja obter um relacionamento um para N entre duas classes, quando se deseja que subclasses possam implementar partes de algoritmos que sejam pertinentes a sua aplicação, quando se quer também evitar duplicação de código, agrupando as subclasses em uma classe única (GURA; CARMO, 2009). No exemplo apresentado no código abaixo tem-se classes que pertencem à um mesmo domínio de aplicação, portanto implementam métodos parecidos, porém cada uma conforme a sua necessidade, não há nenhuma relação entre elas e as duas classes mantem um relacionamento 1:N com a Classe3.

<table>
<tr>
<td><pre>
public class Classe1 {
    ... //Outras implementações
    ArrayList<Classe2> classe = new ArrayList();
    ArrayList<Classe3> classe = new ArrayList();
    ... //Outras implementações
}
</pre></td>
<td><pre>
public class Classe2 {
    ... //Outras implementações
    public void metodo1(){
        //Implementação do método 1
    }
    ... //Outras implementações
}
</pre></td>
<td><pre>
public class Classe3 {
    ... //Outras implementações
    public void metodo1(){
        //Implementação do método 1
    }
    ... //Outras implementações
}
</pre></td>
</tr>
</table>

A solução para o código apresentado utilizando o metapadrão 1:N Connection é transformar as classes Classe2 e Classe3 em subclasses, definindo os métodos em uma superclasse, e cada subclasse implementaria o método da superclasse conforme a sua necessidade.</td></tr>
<tr><td>Solução</td><td>Para solucionar o problema, é criada a classe-gabarito Classe (antiga Classe1) e as classes Classe2 e Classe3 agora foram transformadas em SubClasse1 e SubClasse2, como ilustrado no código abaixo.

<table>
<tr>
<td><pre>
public class Classe {
    ... //Outras implementações
    ArrayList<SuperClasse> superclasse = new ArrayList();

    public void metodo1(){
        //Implementação do método 1
    }
    ... //Outras implementações
}
</pre></td>
<td><pre>
public abstract class SuperClasse extends Classe {
    ... //Outras implementações
    public abstract void metodo1();
    ... //Outras implementações
}
</pre></td>
</tr>
</table>
</td></tr>
</table>

<table>
<tr>
<td></td>
<td>
<table>
<tr>
<td>

```
public class SubClasse1 extends SuperClasse {
   ... //Outras implementações
   public void metodo1(){
      //Implementação do método 1
   }
   ... //Outras implementações
}
```

</td>
<td>

```
public class SubClasse2 extends SuperClasse {
   ... //Outras implementações
   public void metodo1(){
      //Implementação do método 1
   }
   ... //Outras implementações
}
```

</td>
</tr>
</table>

Foi definida também uma classe abstrata como gabarito, a SuperClasse, onde é feita a definição de cada método que as subclasses poderão implementar. Entre a Classe (gabarito) e a SuperClasse (gancho) há uma relação 1 para N (1:N). Com a classe-gabarito e gancho definidas, são definidas as subclasses que irão implementar os métodos definidos na classe-gancho, onde cada subclasse implementa o método conforme a finalidade da sua aplicação.
</td>
</tr>
</table>

Quadro 5 – Refatoração Baseada no Metapadrão 1:N Connection
Fonte: Autoria própria.

<table>
<tr><th>Metapadrão</th><th>1:1 Recursive Unification</th></tr>
<tr><td>Descrição</td><td>O metapadrão 1:1 Recursive Unification pode ser visto como uma derivação do metapadrão Unification, onde a classe-gabarito e gancho são unificadas dentro de uma classe apenas. A chamada do método-gancho é recursiva e referencia apenas um objeto (MATOS 2008).</td></tr>
<tr><td>Problema Endereçado</td><td>O metapadrão 1:1 Recursive Unification deve ser utilizado quando se identifica comportamentos comuns entre subclasses, podendo assim agrupá-los em uma classe para evitar duplicação de código, quando se deseja que cada subclasse implemente seu comportamento e quando há a necessidade de chamar o método-gabarito utilizado recursão (GURA; CARMO, 2009). O exemplo de código utilizado é parecido com o metapadrão 1:1 Unification, onde as classes implementam o mesmo método sendo que a diferença entre as classes é o domínio de aplicação e o comportamento do método, sendo que o método das duas classes utilizam uma parte de implementação por meio da recursividade. Um exemplo de código é apresentado abaixo.

<table>
<tr>
<td><pre>
public class Classe1 {
    ... //Outras implementações
    public void metodo1(){
        //Implementação 1
    }

    public void metodo2(){
        Classe1 classe1 = new Classe1();
        //Implementação 3
        //Implementação utilizando recursividade
    }
    ... //Outras implementações
}
</pre></td>
<td><pre>
public class Classe2 {
    ... //Outras implementações
    public void metodo1(){
        //Implementação 2
    }

    public void metodo2(){
        Classe2 classe2 = new Classe2();
        //Implementação 3
        //Implementação utilizando recursividade
    }
    ... //Outras implementações
}
</pre></td>
</tr>
</table>

No exemplo de código tem-se a Classe1 e a Classe2, as duas classes implementam o mesmo método metodo, porém cada uma com o comportamento voltada para o seu domínio de aplicação, e as duas utilizam recursao em sua implementação.</td></tr>
<tr><td>Solução</td><td>A solução utilizando o metapadrão 1:1 Recursive Unification é definir uma superclasse (gabarito-gancho) que tem uma referência para ela mesma e serve para realizar a recursão, onde as subclasses SubClasse1 (antiga Classe1) e SubClasse2 (antiga Classe2) implementam cada método conforme o seu comportamento. Na SuperClasse existe uma referência para ele mesmo, pois este metapadrão tem a relação 1 para 1. O exemplo do código aplicando o metapadrão é apresentado abaixo.</td></tr>
</table>

```java
public abstract class SuperClasse {
   ... //Outras implementações
   SuperClasse superclasse = new SuperClasse();

   public abstract void metodo1();

   public void metodo2(){
      //Implementação 3
   }
   ... //Outras implementações
}
```

```java
public class SubClasse1 extends SuperClasse {
   ... //Outras  implementações
   @Override
   public void metodo1(){
      //Implementação 1
   }
   ... //Outras implementações
}
```

```java
public class SubClasse2 extends SuperClasse {
   ... //Outras  implementações
   @Override
   public void metodo1(){
      //Implementação 2
   }
   ... //Outras implementações
}
```

Quadro 6 – Refatoração Baseada no Metapadrão 1:1 Recursive Unification
Fonte: Autoria própria.

<table>
<tr><th>Metapadrão</th><th>1:N Recursive Unification</th></tr>
<tr><td>Descrição</td><td>O metapadrão 1:N Recursive Unification se assemelha ao 1:1 Recursive Unification, sendo que a diferença entre eles é que neste padrão a recursão pode referenciar qualquer número de objetos, podendo ter uma lisa de referências (GURA; CARMO, 2009).</td></tr>
<tr><td>Problema Endereçado</td><td>O metapadrão 1:N Recursive Unification deve ser utilizado quando se identifica comportamentos comuns entre subclasses, podendo assim agrupá-los em uma classe para evitar duplicação de código, quando se deseja que cada subclasse implemente seu comportamento e quando há a necessidade de chamar o método-gabarito utilizado recursão (GURA; CARMO, 2009). O metapadrão se assemelha com o 1:1 Recursive Unification, sendo que a única diferença entre eles é que no 1:N Recursive Unification há uma lista de referências. O exemplo de código utilizado é parecido com o metapadrão 1:1 Recursive Unification, onde as classes implementam o mesmo método sendo que a diferença entre as classes é o domínio de aplicação e o comportamento do método. Um exemplo de código é apresentado abaixo.

<table>
<tr>
<td>

```
public class Classe1 {
   ... //Outras implementações
   public void metodo1(){
      //Implementação 1
   }

   public void metodo2(){
      ArrayList<Classe1> classe1 = new ArrayList();
      //Implementação 3
      //Implementação utilizando recursividade
   }
   ... //Outras implementações
}
```

</td>
<td>

```
public class Classe2 {
   ... //Outras implementações
   public void metodo1(){
      //Implementação 2
   }

   public void metodo2(){
      ArrayList<Classe2> classe2 = new ArrayList();
      //Implementação 3
      //Implementação utilizando recursividade
   }
   ... //Outras implementações
}
```

</td>
</tr>
</table>

No exemplo de código tem-se a Classe1 e a Classe2, as duas classes implementam o mesmo método metodo, porém cada uma com o comportamento voltada para o seu domínio de aplicação, e as duas utilizam recursao em sua implementação.</td></tr>
<tr><td>Solução</td><td>A solução utilizando o metapadrão 1:N Recursive Unification é definir uma superclasse (gabarito-gancho) que irá ter uma lista de referências para ela mesma, que serve para realizar a recursão e a definição dos métodos abstratos, onde as subclasses implementam cada método conforme o seu comportamento. Na SuperClasse existe uma lista referência para ele mesmo, pois este metapadrão tem a relação 1 para N. O exemplo do código aplicando o metapadrão é apresentado abaixo.</td></tr>
</table>

<table>
<tr>
<td></td>
<td>

```
public abstract class SuperClasse {
   ... //Outras implementações
   ArrayList<SuperClasse> superclasse = new ArrayList();

   public abstract void metodo1();

   public void metodo2(){
      //Implementação 3
   }
   ... //Outras implementações
}
```

</td>
<td>

```
public class SubClasse1 extends SuperClasse {
    ... //Outras implementações
    @Override
    public void metodo1(){
        //Implementação 1
    }

    @Override
    public void metodo2(){
        super.metodo2();
    }
    ... //Outras implementações
}
```

</td>
<td>

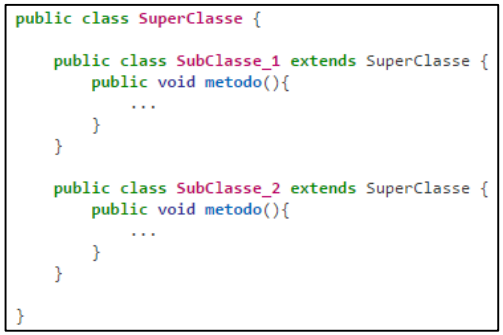
```
public class SubClasse2 extends SuperClasse {
    ... //Outras implementações
    @Override
    public void metodo1(){
        //Implementação 2
    }

    @Override
    public void metodo2(){
        super.metodo2();
    }
    ... //Outras implementações
}
```

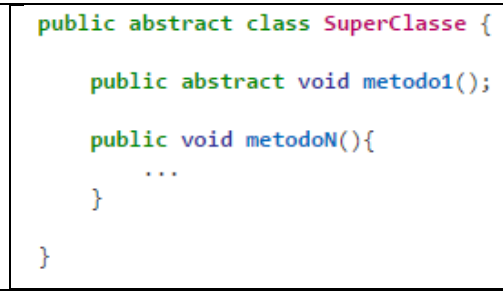
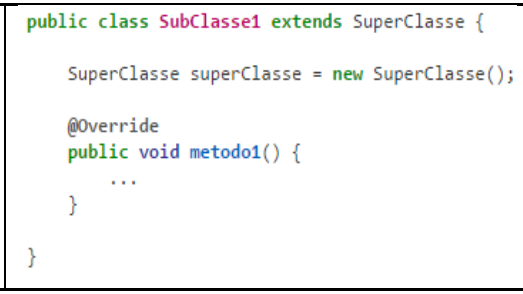
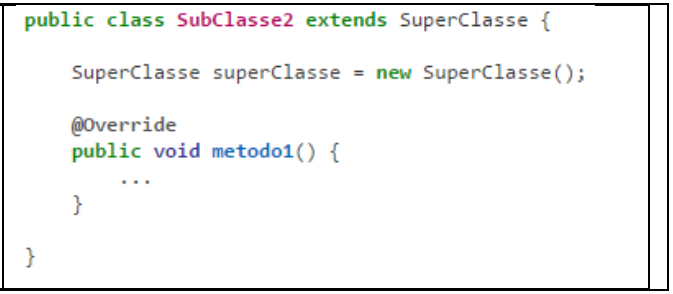
</td>
</tr>
</table>

Quadro 7 – Refatoração Baseada no Metapadrão 1:N Recursive Unification
Fonte: Autoria própria

<table>
<tr><th>Metapadrão</th><th>1:1 Recursive Connection</th></tr>
<tr><td>Descrição</td><td>O metapadrão 1:1 Recursive Connection faz com que um objeto da classe-gabarito refira-se a exatamente uma instância da classe-gancho, sendo que a classe-gabarito é descendente de suas classes-ganchos (MATOS 2008).</td></tr>
<tr><td>Problema Endereçado</td><td>O metapadrão 1:1 Recursive Connection deve ser implementado quando há a necessidade de se chamar um método-gabarito várias vezes, quando se quer referenciar apenas um objeto e separar comportamentos específicos nas subclasses (GURA; CARMO, 2009). Um exemplo de código da aplicação deste metapadrão é apresentado abaixo.

O código apresentado contém uma superclasse com duas subclasses internas. As subclasses poderiam ser retiradas das superclasses, sendo implementadas através de herança e a superclasse manteria a referência para um objeto das subclasses.</td></tr>
<tr><td>Solução</td><td>A solução utilizada para o problema foi retirar as subclasses internas da superclasse, porém as subclasses continuam estendendo a superclasse. Agora na subclasse também há a referência para um objeto da superclasse. Na solução também é definido um método abstrato na superclasse e as subclasses ficam responsável por sobrescrevê-lo e realizar as chamadas recursivamente. O código com a implementação do metapadrão é apresentado abaixo.</td></tr>
</table>

Quadro 8 – Refatoração Baseada no Metapadrão 1:1 Recursive Connection
Fonte: Autoria própria.

<table>
<tr><th>Metapadrão</th><th>1:N Recursive Connection</th></tr>
<tr><td>Descrição</td><td>O metapadrão 1:N Recursive Connection faz com que um objeto de uma classe-gabarito tenha uma lista de referência para a classe-gancho. A implementação do metapadrão é realizada a partir de herança e por método abstrato que referencia apenas um objeto. O metapadrão se assemelha ao 1:1 Recursive Connection, sendo a única diferença a quantidade de referências que podem existir entre as classes-gabarito e gancho (MATOS 2008).</td></tr>
<tr><td>Problema Endereçado</td><td>O metapadrão 1:N Recursive Connection deve ser implementado quando há a necessidade de se chamar um método-gabarito várias vezes, quando se quer referenciar uma lista de objetos e separar comportamentos específicos nas subclasses (GURA; CARMO, 2009). Um exemplo de código da aplicação do metapadrão é apresentado abaixo.

<pre>
public class SuperClasse {

    public class SubClasse_1 extends SuperClasse {
        public void metodo(){
            ...
        }
    }

    public class SubClasse_2 extends SuperClasse {
        public void metodo(){
            ...
        }
    }

}
</pre>

O código apresentado contém uma superclasse com duas subclasses internas. As subclasses poderiam ser retiradas das superclasses, sendo implementadas através de herança e a superclasse manteria a referência para uma lista de objetos das subclasses.</td></tr>
<tr><td>Solução</td><td>A solução utilizada para o problema foi retirar as subclasses internas da superclasse, porém as subclasses continuam estendendo a superclasse. Agora na subclasse também há uma lista de referências para objetos da superclasse. Na solução também foi definido um método abstrato na</td></tr>
</table>

superclasse e as subclasses ficam responsável por sobrescrever esse método e realizar as chamadas recursivamente. O código com a implementação do metapadrão é apresentado abaixo.

```
public abstract class SuperClasse {

    public abstract void metodo1();

    public void metodoN(){
        ...
    }

}
```

```
public class SubClasse1 extends SuperClasse {

    List<SuperClasse> superClasse = new List<SuperClasse>();

    @Override
    public void metodo1() {
        ...
    }

}
```

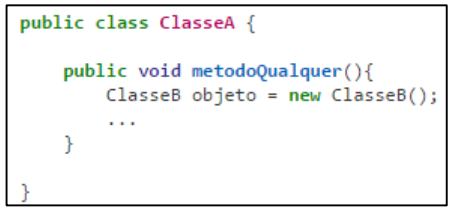
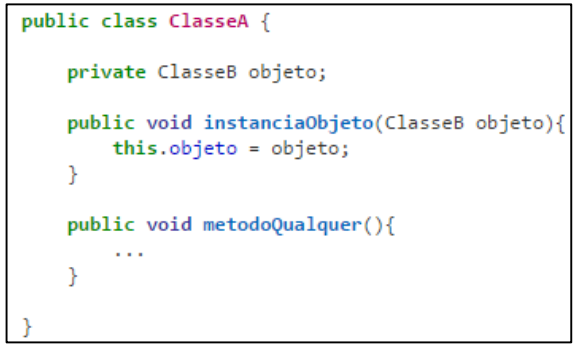
```
public class SubClasse2 extends SuperClasse {

    List<SuperClasse> superClasse = new List<SuperClasse>();

    @Override
    public void metodo1() {
        ...
    }

}
```

Quadro 9 – Refatoração Baseada no Metapadrão 1:N Recursive Connection
Fonte: Autoria própria.

<table>
<tr><th>Característica</th><th>Inversão de Controle</th></tr>
<tr><td>Descrição</td><td>A Inversão de Controle é uma maneira de tratar instâncias de uma classe, fazendo isso externamente, e não instanciando um objeto diretamente dentro da classe.</td></tr>
<tr><td>Problema Endereçado</td><td>A Inversão de Controle deve ser usada quando se tem uma instanciação de objetos de uma determinada classe B dentro da classe A, como segue o exemplo do código abaixo.
 
Neste exemplo a ClasseA cria uma instância da ClasseB dentro do método metodoQualquer, não há nenhum erro até então, porém se a classe ClasseB for modificada, a classe ClasseA e todas as outras classes que instanciam a ClasseB sofreriam com a modificação e cada instância deveria ser modificada, sendo assim, há um forte acoplamento.</td></tr>
<tr><td>Solução</td><td>Com a Inversão de Controle não há mais uma instanciação direta de objetos e as classes passam a receber uma referência do objeto que já foi instanciado em outra classe, assim não há mais a dependência da ClasseA com a ClasseB, como pode ser visto no código abaixo.
 
Neste exemplo a classe ClasseA recebe uma instância da classe ClasseB a partir do método instanciaObjeto, e atribui essa instância para o objeto criado na classe ClasseA, assim qualquer modificação realizada na ClasseB não será necessário modificar a classe ClasseA.</td></tr>
</table>

Quadro 10 – Refatoração Baseada em Inversão de Controle
Fonte: Autoria própria.

Após as refatorações serem aplicadas a partir de uma das técnicas possíveis, é realizada uma nova verificação do sistema retornando ao passo *Utilizar a ferramenta* seguindo para o passo de *Verificar bad smells do projeto*. Com a nova análise da ferramenta é possível verificar se ainda há *bad smells* que necessitam de refatoração. Se houver, é aplicado novamente o passo de *Aplicar técnicas* para tentar solucionar os *bad smells* que ainda estão presentes. Caso não haja mais *bad smells* ou soluções a serem implementadas, segue-se para o passo *Validar o sistema*, que é realizado a partir de testes nas funcionalidades do sistema, verificando se ele continua com o mesmo comportamento de antes das refatorações serem aplicadas, podendo ser usado como base a primeira etapa *Entender o sistema*, onde é feito a utilização do sistema para verificar como ele se comporta.

Outra forma de se validar o sistema é utilizando o conjunto de entradas e saídas que é proposto por Rapeli, que consiste em realizar uma catalogação da entradas e saídas do sistema antes da refatoração ser aplicada, e após a refatoração é realizado as mesmas entradas e se espera obter as mesmas saídas. É importante ressaltar que o método proposto não tem como foco a fase de teste, isto pode ser realizado em um trabalho futuro.

Quando o sistema for verificado e validado, garantindo que seu comportamento não se modificou após a refatoração, volta-se para a iteração dos módulos, onde será selecionado o próximo módulo a ser refatorado. Ao final de todos os módulos, o processo de refatoração chega ao seu fim. O método proposto não obriga que o desenvolvedor refatore todos os módulos, mas é flexível por permitir que ele escolha quais deseja refatorar.

5 RESULTADOS

Este capítulo apresenta os resultados obtidos a partir da aplicação do método de refatoração proposto. A seção 5.1 descreve o uso do método para realizar a refatoração do Framework de Preço de Venda (FrameMK). A seção 5.2 apresenta algumas estatísticas referentes ao uso do método de refatoração. A seção 5.3 faz uma análise comparativa do método com os de Rapeli (2006) e Mens e Tourwé (2004).

5.1 APLICAÇÃO DO MÉTODO PROPOSTO

Esta seção apresenta os resultados obtidos da aplicação do método no Framework de Preço de Venda (FrameMK). A seção 5.1.1 descreve os resultados obtidos do primeiro passo da etapa *Entender o Sistema*. A seção 5.1.2 descreve os resultados atingidos por meio da execução do segundo passo da etapa *Entender o Sistema*. A seção 5.1.3 apresenta os resultados alcançados por meio do terceiro passo da etapa *Entender o Sistema*. A seção 5.1.4 relata os resultados obtidos da etapa de *Ordenar módulos*. A seção 5.1.5 descreve os resultados alcançados da terceira etapa de *Refatorar módulo*.

5.1.1 ENTENDER O SISTEMA – PASSO: UTILIZAR O SISTEMA

A primeira etapa do método de refatoração proposto é *Entender o Sistema*. O sistema utilizado como estudo de caso para a aplicação do método é o FrameMK (descrito na seção 3.4). O primeiro passo dessa etapa foi utilizar o sistema por meio de sua execução. A instalação do FrameMK necessitou de 3 (três) software: Eclipse Luna (ECLIPSE, 2015), Firebird 2.5 (FIREBIRD, 2015) e Tomcat 7.0 (2015). A *IDE Eclipse* permite abrir o projeto do framework. O *Firebird* é o banco de dados utilizado pelo framework. *Tomcat Apache* é o servidor da aplicação.

O FrameMK contém uma página de início que contém uma explicação sobre sua finalidade. A partir desta página é possível visualizar os trabalhos já

desenvolvidos no FrameMK, os créditos para os trabalhos desenvolvidos e contato com os desenvolvedores. Para poder entrar no sistema é necessário ter um cadastro, que pode ser realizado a partir da opção *"Cadastrar-se"*. Caso já haja um cadastro efetuado, o usuário pode entrar no sistema por meio da opção *Login*. A página inicial do FrameMK já foi apresentada na figura 2 da seção 3.4.

Assim que o usuário entra no sistema, aparecem outras 5 (cinco) opções de operações no framework, que são os métodos para o cálculo do preço de venda.

Cada método de cálculo de preço de venda possui a sua particularidade, pois cada método implementa um tipo de cálculo diferente. A figura 9 ilustra como é o sistema após o *login* ser realizado.

Figura 9 - Login realizado no FrameMK
Fonte: Autoria própria

Como cada método possui a sua particularidade, o usuário deve escolher qual deseja usar. Como o intuito do passo *Utilizar o Sistema* é apenas o desenvolvedor conhecer a aplicação, logo todos os métodos não serão explorados, neste trabalho será mostrado o uso do método Custo Pleno para calcular o preço de venda de um produto.

Quando selecionado o método do Custo Pleno outras duas opções aparecem para o usuário, como exibido na figura 10.

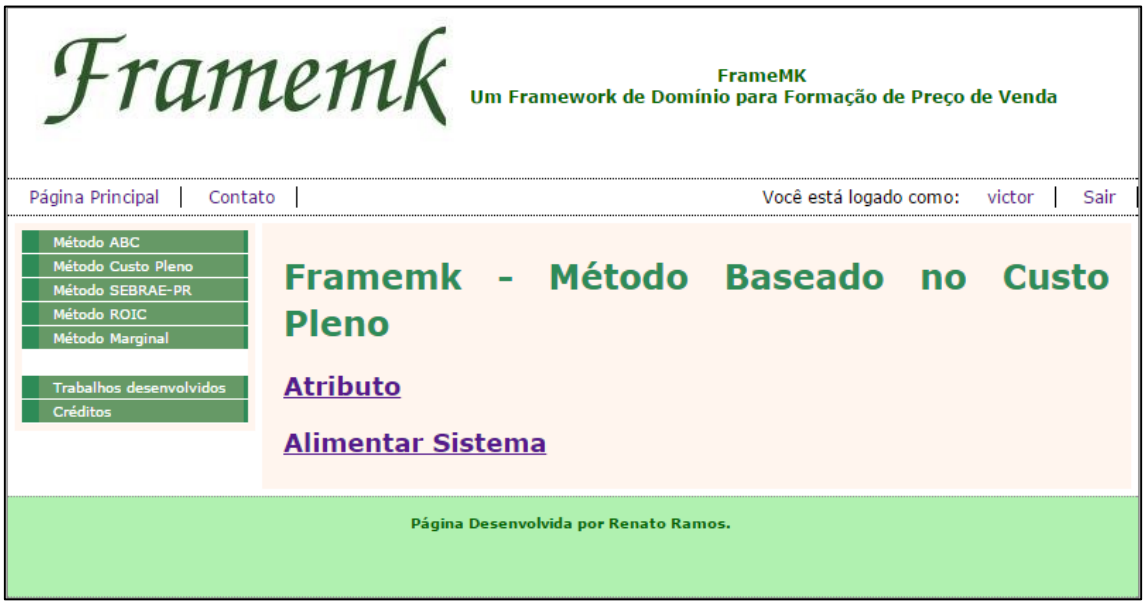


Figura 10 - Método do Custo Pleno
Fonte: Autoria própria

Ao selecionar a opção “Atributo”, o framework apresenta outra página que mostra quais são os atributos que serão utilizados para realizar o cálculo do preço de venda. Os atributos que já estiverem cadastrados podem ser editados e/ou desativados. A figura 11 mostra como é apresentada a página dos atributos.

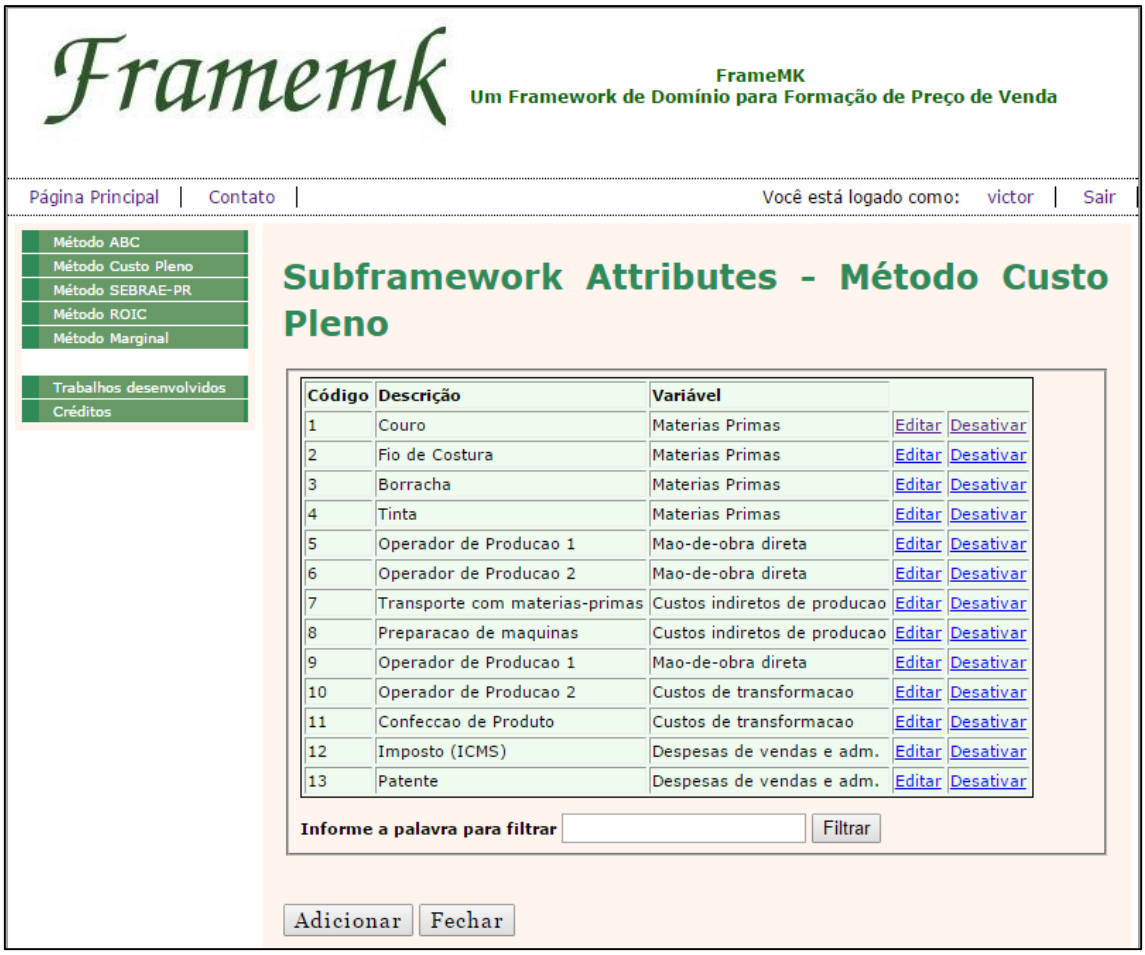


Figura 11 - Atributos do Método Custo Pleno
Fonte: Autoria própria

Também há a opção por filtrar algum atributo utilizando uma palavra-chave. Quando algo é filtrado, somente é exibido o(s) atributo(s) que contém a palavra filtrada, como exibe a figura 12.

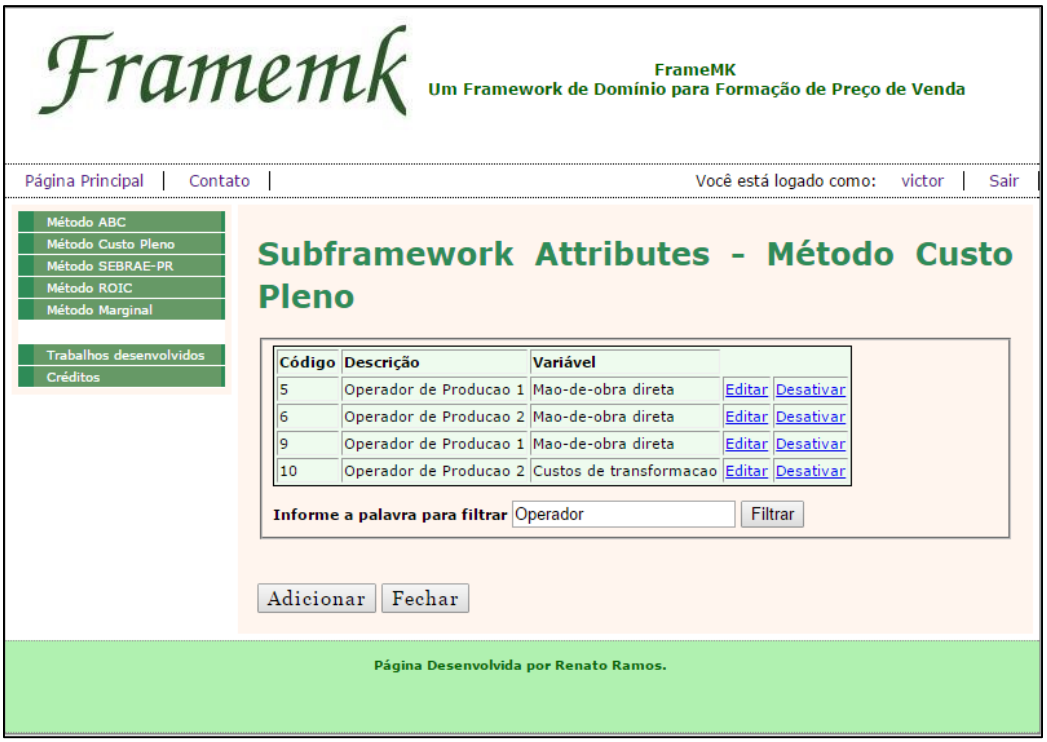


Figura 12 - Filtro por atributo
Fonte: Autoria própria

Outra opção é inserir um novo atributo, que pode ser realizada a partir o botão “Adicionar”. Ao clicar no botão uma nova tela irá aparecer para poder inserir o nome do atributo e qual a sua variável, onde as variáveis já são definidas pelo sistema. A figura 13 ilustra como ocorre a inserção de um atributo.

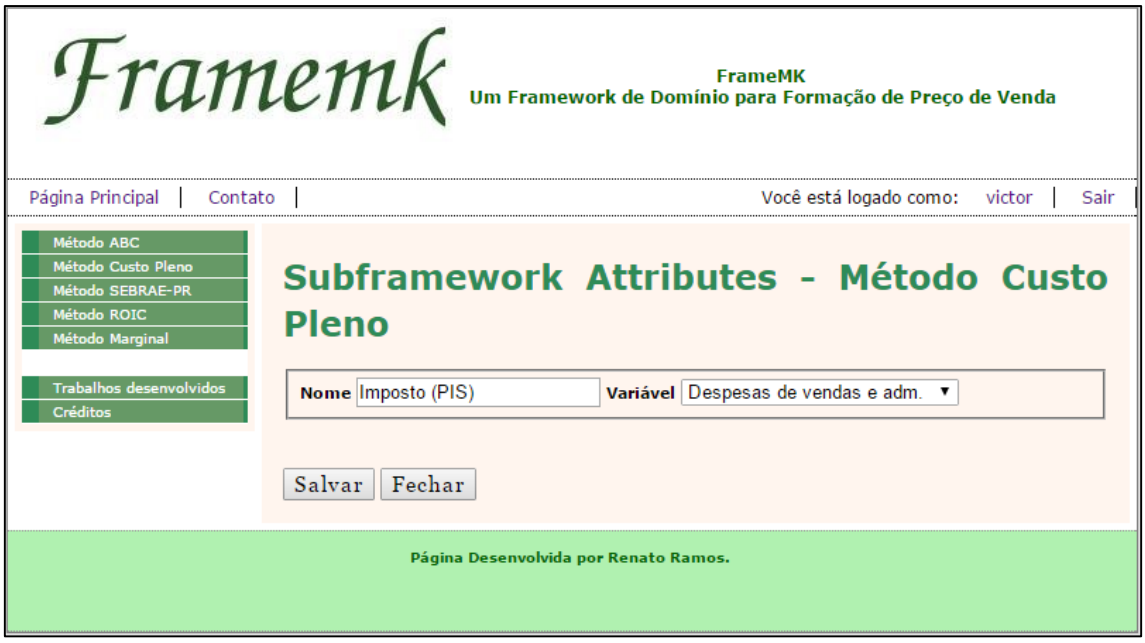
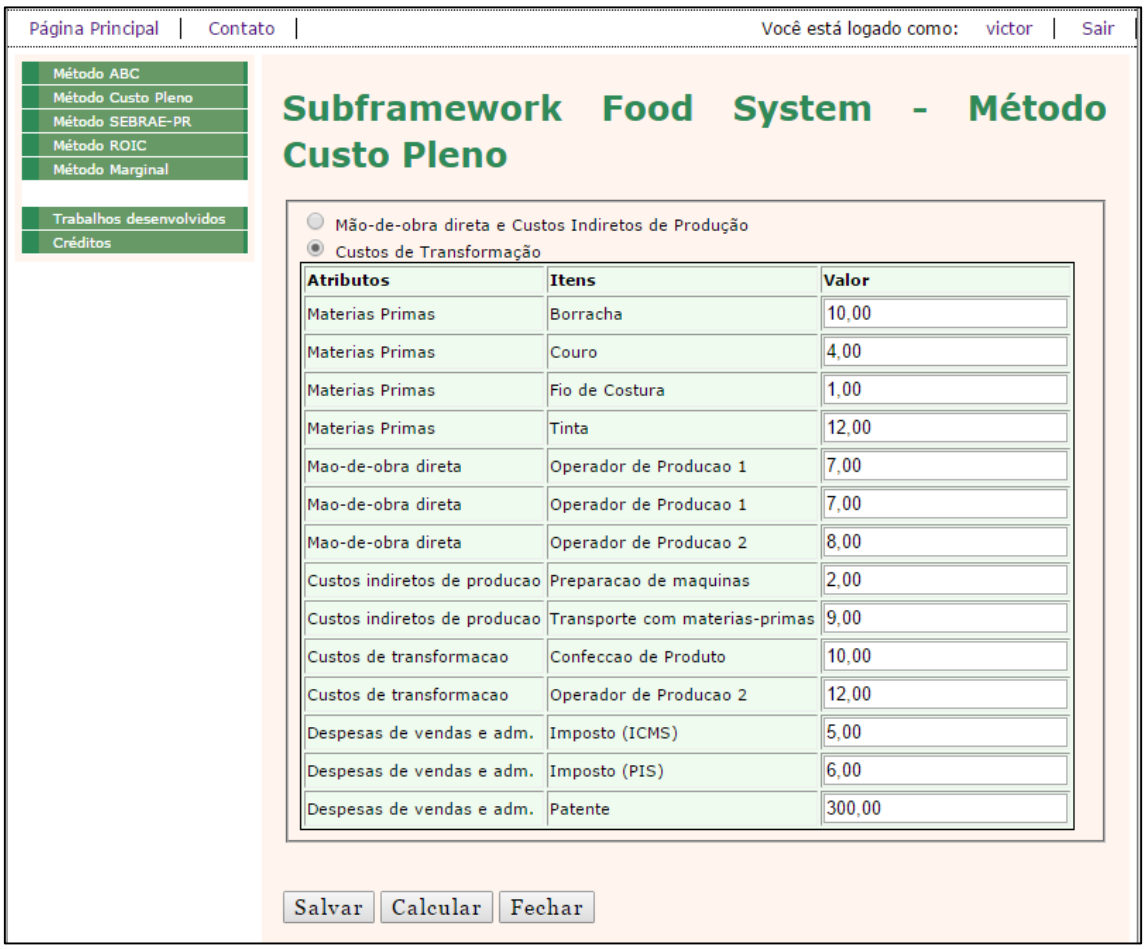


Figura 13 - Inserção de um atributo
Fonte: Autoria própria

Após todos os atributos estarem cadastrados, pode-se acessar a opção de *Alimentar Sistema* (figura 14). Ao clicar nesta opção, é possível escolher entre duas opções: *Mão-de-obra direta e Custos Indiretos de Produção* ou *Custos de Transformação*. É possível também atribuir valores para os atributos cadastrados no FrameMK.



Atributos	Itens	Valor
Materias Primas	Borracha	10,00
Materias Primas	Couro	4,00
Materias Primas	Fio de Costura	1,00
Materias Primas	Tinta	12,00
Mao-de-obra direta	Operador de Producao 1	7,00
Mao-de-obra direta	Operador de Producao 1	7,00
Mao-de-obra direta	Operador de Producao 2	8,00
Custos indiretos de producao	Preparacao de maquinas	2,00
Custos indiretos de producao	Transporte com materias-primas	9,00
Custos de transformacao	Confeccao de Produto	10,00
Custos de transformacao	Operador de Producao 2	12,00
Despesas de vendas e adm.	Imposto (ICMS)	5,00
Despesas de vendas e adm.	Imposto (PIS)	6,00
Despesas de vendas e adm.	Patente	300,00

Figura 14 - Alimentar Sistema do Método de Custo Pleno
Fonte: Autoria própria

Após os valores serem configurados, deve-se clicar no botão *Salvar* para que as alterações realizadas sejam armazenadas. Para realizar o cálculo do preço de venda a partir dos atributos configurados deve-se clicar no botão *Calcular*, onde uma nova página será aberta para informar qual a margem de lucro desejado sobre o custo do produto. Ao informar o lucro e clicar no botão *Calcular* o preço do produto será apresentado no campo *Resultado R$*. O processo para o cálculo de preço de venda é apresentado na figura 15.

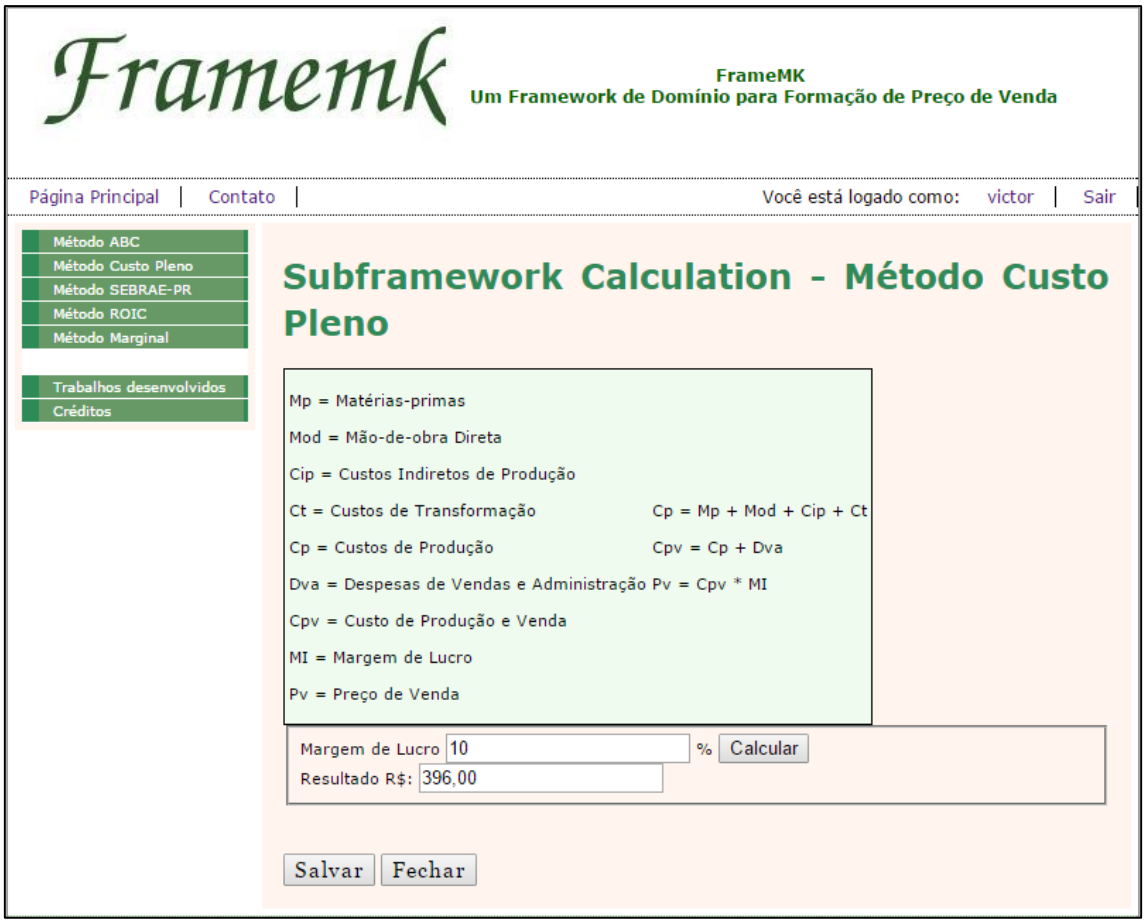


Figura 15 - Cálculo do preço de venda
Fonte: Autoria própria

Para finalizar o cálculo é necessário pressionar o botão *Salvar* para que os dados sejam armazenados.

5.1.2 ENTENDER O SISTEMA – PASSO: ORDENAR MÓDULOS

O FrameMK é dividido em 5 módulos apresentados na figura 16.

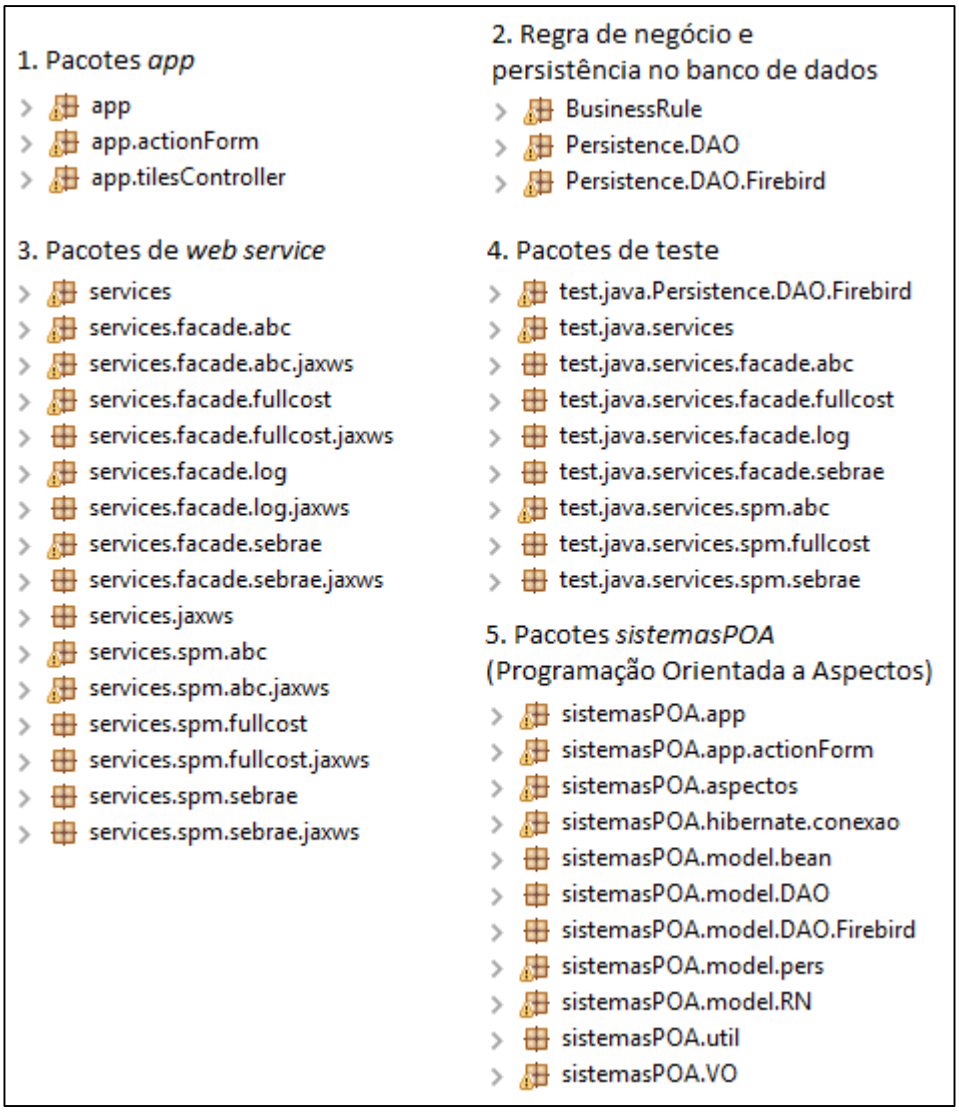


Figura 16 - Módulos que compõem o FrameMK
Fonte: Autoria própria

O primeiro módulo trabalha com a parte visual do FrameMK, sendo ele composto pelos pacotes *app*, *app.actionForm* e *app.tilesController*.

O segundo módulo tem como objetivo a persistência no banco de dados, que é feita pelos pacotes *Persistence*, e cuida também da regra de negócio do *framework*, por meio do pacote *BusinessRule*.

O terceiro módulo é composto pelos pacotes *services*, que contém a implementação do serviço web do FrameMK.

O quarto módulo é formado pelos pacotes *test*, sendo o local em que se tem as classes de teste do sistema web.

O quinto módulo é composto pelo conjunto de pacotes sistemasPOA, onde é implementado o sistema de *login*, sendo esse implementado utilizando orientação a aspectos.

5.1.3 Entender o Sistema – Passo: Gerar Diagrama

O terceiro passo da etapa de Entender o Sistema é a geração do diagrama de classe para que se tenha uma forma visual e mais fácil de se entender como as classes se relacionam. Para o FrameMK foi gerado o diagrama de classes para o módulo *app*, sendo feito para dois pacotes, o *app.actionForm* e o *app*, que contém classes que se relacionam. O diagrama foi gerado através da ferramenta *Enterprise Architect* (ENTERPRISE ARCHITECT, 2015), que gerar automaticamente o diagrama a partir do código do sistema. O diagrama é ilustrado na figura 17.

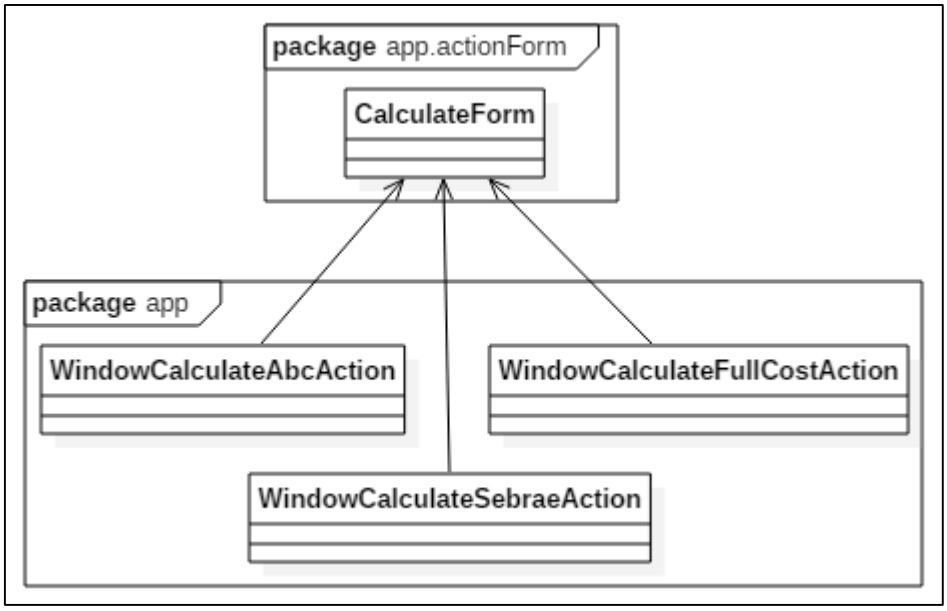


Figura 17 - Diagrama de Classe do relacionamento do pacote *app* com o pacote *app.actionForm*
Fonte: Autoria própria

As classes utilizadas para a geração do diagrama foram a *CalculateForm* que pertence ao pacote *app.actionForm* e as classes *WindowsCalculateAbcAction*, *WindowCalculateFullCostAction* e *WindowCalculateSebraetAction*.

A partir do diagrama é possível observar que as classes *WindowsCalculateAbcAction*, *WindowCalculateFullCostAction* e *WindowCalculateSebraeAction* possui uma referência para a classe *CalculateForm*, sendo assim, são grandes as chances de que se algo for modificado dentro da classe *CalculateForm*, as classes que mantém uma

referência para a *CalculateForm* sofrerão com a alteração, assim deve-se ter uma atenção ao refatorar a classe *CalculateForm*.

Assim como o relacionamento entre as classes exibidos no digrama da figura 17, existem outros diagramas possíveis de serem gerados no estudo de caso FrameMK, porém por serem muitos, tais diagramas não serão apresentados nesse trabalho.

O diagrama facilitou a compreensão do relacionamento entre as classes, sendo possível observar o impacto da refatoração.

5.1.4 ORDENAR MÓDULOS

A segunda etapa do processo de refatoração tem como objetivo ordenar os módulos do sistema em ordem crescente pela quantidade de *bad smells* para posteriormente serem usados no processo de refatoração. A quantidade de *bad smells* do sistema pode ser obtida por meio de ferramentas. No processo de refatoração do FrameMK se utilizou a ferramenta *SonarQube*, devido ao conhecimento prévio de como utilizar a ferramenta, ser gratuita e se adequar as necessidades da refatoração.

Através da análise do FrameMK pela ferramenta SonarQube se tem um panorama geral de como o framework se encontra. Este panorama é de todo o FrameMK e não apenas de um módulos específico, tal panorama é mostrado na figura 18.

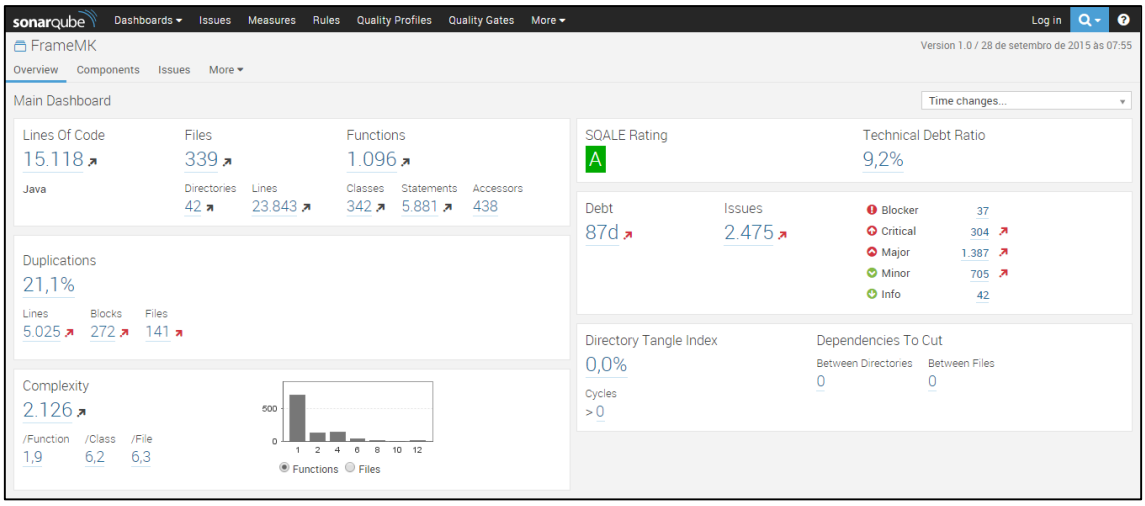


Figura 18 - Panorama geral do FrameMK através da ferramenta SonarQube
Fonte: Autoria própria

A figura 18 contém algumas informações relevantes para a refatoração do FrameMK. É possível observar que o framework possui 342 classes, com 1.096 funções juntamente com mais de 15 mil linhas de código, sendo que há 21,1% de duplicação, em 5.025 linhas de código. Referente a parte específica da refatoração, a ferramenta *SonarQube* mostra que há 2.475 refatoração possíveis, sendo elas subdivididas em 5 categorias:

1. *Blocker* (37): esta categoria corresponde aos locais do código que merecem maior atenção, sendo o grau mais emergencial de refatoração necessária. Um exemplo deste grau de refatoração é a remoção de *returns* dentro de blocos *finally*.
2. *Critical* (304): são o segundo nível de refatoração mais importantes no sistema, um dos exemplos de código referente a este tipo de refatoração crítica é o tratamento de erros e exceções que o sistema faz.
3. *Major* (1.387): é o nível de refatoração que possui mais lugares no código a serem refatorados. Exemplo deste tipo de refatoração é a utilização de *ArrayList* ou *List* no lugar de *Vector*, pois o Vector caiu em desuso devido ao seu desempenho inferior perante o *ArrayList* e o *List*.
4. *Minor* (705): são refatorações de menor importância para o sistema, como por exemplo a renomeação de variáveis conforme a convenção da linguagem de programação utilizada, neste caso Java.

5. *Info* (42): são refatorações que não afetam de fato o funcionamento do sistema, como por exemplo, a remoção de comentários em lugares que não são necessários.

A ordenação dos módulos foi obtida utilizando o algoritmo de ordenação apresentado na etapa 2 do método de refatoração. Tal algoritmo foi implementado na linguagem C, e recebia como entrada o nome do módulo e a quantidade de *bad smells* que ele continha, ao fim da entrada, é digitado a leta “S”, para informar que a entrada de dados havia sido concluída. A saída do algoritmo é a classificação do módulo, seu nome e a quantidade de *bad smells* que ele contém, como ilustrado na figura 19.

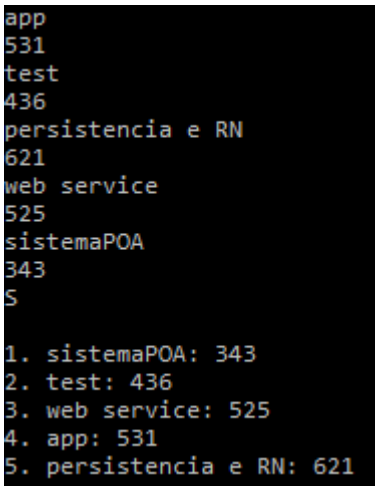


Figura 19 - Algoritmo de ordenação dos módulos
Fonte: Autoria própria

Pelo resultado do algoritmo é possível observar que todos os 5 (cinco) módulos contém uma grande quantidade de *bad smell.* Outra informação também é que o último módulo da classificação contém quase o dobro de bad smells presentes no primeiro módulo da classificação.

5.1.5 REFATORAR O SISTEMA

A terceira etapa do método é onde de fato ocorre a refatoração do sistema. Nesta etapa utiliza-se os módulos em ordem crescente obtidos na etapa 2.

Essa etapa inicia com a iteração de todos os módulos, verificando se eles serão ou não refatorados. O primeiro módulo classificado em ordem

crescente foi o *sistemaPOA*, porém este módulo não será refatorado nesse trabalho, o segundo módulo da ordem foi o *test* o qual também não será refatorado, próximo módulo classificado foi o *web service* assim como os dois primeiros também não será refatorado. O quarto módulo é o *app* e será refatorado, logo, na iteração utiliza-se a ferramenta novamente, porém agora com o foco no módulo e na refatoração.

O único módulo que será refatorado nesse trabalho é o *app*, pois como o Grupo de Pesquisa de Sistemas de Informação (GPSI), para o qual este trabalho está sendo realizado, contém outros integrantes que também realizam pesquisas sobre refatoração, eles irão realizar a refatoração dos outros módulos que não foram refatorados.

A primeira refatoração a ser realizada a partir da sugestão da ferramenta está relacionada ao objeto *atribute* quando é nulo (*null*) utilizando o método *equals*, como ilustra a figura 20.

```
if (attribute.equals(null)) {
    attribute = new AbcAttribute();
    validates = new ValidatesAbc();
}
```

Figura 20 - Comparação utilizando o método *equals*
Fonte: Autoria própria

A utilização do método *equals* juntamente com *null* não é recomendado pois pode acontecer do sistema apresentar o erro de *null pointer exception*. Para solucionar este problema foi utilizada a sugestão da ferramenta *SonarQube*, que consiste em fazer a comparação direta do objeto *attribute* com *null*, como mostrado na figura 21.

```
if (attribute == null) {
    attribute = new AbcAttribute();
    validates = new ValidatesAbc();
}
```

Figura 21 – Comparação direta do objeto com *null*
Fonte: Autoria própria

Outra refatoração necessária quando se utiliza o método *equals* é na comparação com *strings*, onde é recomendado a utilização da *string* que se quer comparar do lado esquerdo da expressão e o no método *equals* ser passado o objeto para comparação. Este tipo de refatoração previne que o erro *null pointer*

exception ocorra e também elimina a necessidade de verificar se o objeto a ser comparado é *null*. A figura 22 mostra o exemplo de código do módulo que faz a comparação não recomendada com uma *string*.

```
if (result.equals("cadastrado com sucesso"))
```

Figura 22 - Comparação usando *equals* juntamente com *String*
Fonte: Autoria própria

A solução para este problema é inverter a *string* com o objeto *result*, prevenindo o erro de *null pointer exception* caso o objeto *result* seja nulo. A figura 23 mostra a solução para o problema.

```
if ("cadastrado com sucesso".equals(result))
```

Figura 23 - Solução para o problema da utilização de *equals* com S*tring*
Fonte: Autoria própria

Dentro do módulo *app* existem outros 43 problemas iguais a esses dois apresentados, tais problemas foram resolvidos da mesma maneira, logo não serão apresentados detalhes neste trabalho.

Outra refatoração necessária apresentada pela ferramenta é voltada para o aumento de desempenho da aplicação, ou seja, substituição de *Vector* por *ArrayList*. A figura 24 mostra um exemplo do código do FrameMK onde é possível realizar esta substituição.

```
Vector precos = abcProductBR.calculate(abcProductionLine.getCode());
```

Figura 24 - Utilização de *Vector*
Fonte: Autoria própria

Para solucionar este problema foi realizada a substituição de *Vector* por *ArrayList*, como mostrado na figura 25.

```
ArrayList precos = abcProductBR.calculate(abcProductionLine.getCode());
```

Figura 25 - Utilização de *ArrayList*
Fonte: Autoria própria

Devido à maneira diferente de se trabalhar com *Vector* e *ArrayList*, foi necessário a alteração em outro local do código onde era requisitado o valor da lista em determinado índice, para tal, era utilizado *precos.elementAt(count)*, onde

o método *elementAt* retorna o valor que está na lista *precos* no índice *count*, como mostrado na figura 26.

```
div[count] = (Double) precos.elementAt(count)
        / unid_produzidas[count];
```

Figura 26 - Utilização do método *elementAt*
Fonte: Autoria própria

A solução foi a alteração do método *elementAt(count)* para o *get(count)*, que tem a mesma finalidade de trazer o conteúdo da lista em determinado índice, porém o *get* é utilizado para *ArrayList*, o código da solução é apresentado na figura 27.

```
div[count] = (Double) precos.get(count)
        / unid_produzidas[count];
```

Figura 27 - Utilização do método *get*
Fonte: Autoria própria

Esta refatoração afeta também outras classes e módulos que não estão no conjunto *app*, pois na instanciação de *precos* é utilizado um objeto do pacote *BusinessRule*, logo para o sistema manter o seu funcionamento, foi necessário modificar a classe do objeto utilizado. Esta modificação foi realizada em duas partes do código dentro da classe. A primeira foi na declaração, onde também foi alterado o tipo *Vector* para *ArrayList*, como mostra a figura 28.

```
public ArrayList calculate(int productionLineCode) throws Exception {
```

Figura 28 - Modificação da declaração do método para ArrayList
Fonte: Autoria própria

O segundo lugar do código que foi modificado foi o tipo da lista *valores*, sendo necessário a sua alteração de *Vector* para *ArrayList*, conforme a figura 29.

```
ArrayList valores = new ArrayList();
```

Figura 29 - Alteração do tipo da lista valores
Fonte: Autoria própria

Esta modificação se fez necessária pois o retorno do método é a lista *valores*, sendo que o tipo do retorno deve ser o mesmo da declaração do método.

Assim como a refatoração mostrada anteriormente, existem outros 69 trechos de código que devem ser modificados de *Vector* para *ArrayList*, sendo

que a forma de se realizar a refatoração foi a mesma, portanto não serão apresentados os detalhes.

Foram refatoradas também outros problemas além dos apresentados utilizando a ferramenta SonarQube, porém tais refatoração não serão explicadas o passo a passo de como foram realizadas. O quadro 11 mostra qual era as refatorações necessárias, suas classes e quais foram as soluções.

Problema	Classes	Solução
Utilização excessivo de parênteses não necessários em determinadas expressões.	*MainFramemkAction; WindowAddAbcAction WindowAddActivityAction; WindowAddFullCostAction; WindowAddProductAction; WindowAddProductionLineAction WindowAddSebraeAction; WindowCalculateAbcAction WindowCalculateFullCostAction; WindowCalculateSebraeAction WindowFindAbcAction; WindowFindActivityAction WindowFindFullCostAction; WindowFindProductAction WindowFindProductionLineAction; WindowFindSebraeAction; WindowFoodSystemAbcAction; WindowFoodSystemFullCostAction; WindowFoodSystemSebraeAction WindowMenuAbcAction; WindowMenuFullCostAction WindowMenuMarginalCostAction; WindowMenuSebraeAction*	A solução foi a remoção dos parênteses excessivos que poderiam vir a confundir o entendimento da expressão.
Variáveis declaradas, porém, não são utilizadas.	*MainFramemkAction; WindowAddAbcAction WindowAddActivityAction; WindowAddFullCostAction WindowAddProductAction; WindowAddProductionLineAction*	A solução foi remover a declaração das variáveis não utilizadas.
Blocos de código comentados.	*MainFramemkAction; WindowAddAbcAction WindowAddActivityAction; WindowAddFullCostAction WindowAddProductAction; WindowAddProductionLineAction WindowAddSebraeAction*	A solução foi retirar os códigos existentes no projeto que estavam comentados.
Classes com o mesmo nome de interfaces que implementam.	*WindowCalculateFullCostAction WindowFoodSystemAbcAction WindowFoodSystemFullCostAction WindowFoodSystemSebraeAction*	A solução utilizada foi renomear o nome das classes para que elas não tenham o mesmo nome das interfaces que elas implementam.
Comparação de instancias tipo de dados *float* com 0.	*WindowCalculateSebraeAction*	A solução foi utilizar a função *Float.floatToRawIntBits()* que faz uma comparação precisa utilizando o número de *bits*. Essa comparação com *float* é necessária devido ao framework ser de cálculo de preço de venda de produtos e/ou serviços, sendo assim os valores dos cálculos devem ser exatos e não terem margem para erros.

Quadro 11 – Outras refatorações realizadas a partir da ferramenta no FrameMK
Fonte: Autoria própria

Após realizar as refatorações sugeridas pela ferramenta, o próximo passo foi analisar o código e verificar se existe a possibilidade de aplicação: das técnicas de refatoração de Fowler (1999), metapadrões, inversão de controle e padrões de projeto.

Ao analisar o código, iniciou-se a aplicação da refatoração por meio do conceito de metapadrões apresentados na seção 4.1. Um dos metapadrões que foi utilizado na refatoração é o *Unification*. Este metapadrão foi utilizado pois haviam 3 classes dentro do módulo *app* que utilizavam o mesmo método, sendo que a diferença entre esses métodos era a forma de realizar o cálculo de preço de venda, salvar e fechar a página de cálculo.

As 3 classes identificadas utilizavam ao todo 3 métodos idênticos, sendo eles o *calculate*, *save* e *close*, como mostrado na figura 30.

```
public ActionForward calculate(ActionMapping mapping,
                               ActionForm form,
                               HttpServletRequest request,
                               HttpServletResponse response)
                                       throws Exception {
public ActionForward save(ActionMapping mapping,
                          ActionForm form,
                          HttpServletRequest request,
                          HttpServletResponse response)
                                  throws Exception {
public ActionForward close(ActionMapping mapping,
                           ActionForm form,
                           HttpServletRequest request,
                           HttpServletResponse response)
                                   throws Exception {
```

Figura 30 – Métodos *calculate*, *save* e *close*
Fonte: Autoria própria

A solução utilizada foi criar a classe abstrata *WindowCalculateUnification*, definindo dentro dessa classe os 3 métodos e em cada classe que utiliza esses métodos estender da nova classe criada. A figura 31 mostra a criação da nova classe.

```
public abstract class WindowCalculateUnification extends LookupDispatchAction  {

    public abstract ActionForward calculate(ActionMapping mapping,
                                    ActionForm form,
                                    HttpServletRequest request,
                                    HttpServletResponse response)
                                         throws Exception;

    public abstract ActionForward close(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
                                        throws Exception;

    public abstract ActionForward save(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
                                        throws Exception;

    protected abstract Map getKeyMethodMap();

}
```

Figura 31 - Criação da classe *WindowCalculateUnification*
Fonte: Autoria própria

Com a classe criada e os métodos definidos, a última alteração necessária no código é fazer as subclasses estendam esta superclasse, adicionando o código *extends WindowCalculateUnification* na declaração de cada uma das 3 subclasses, como mostrado na figura 32.

```
public class WindowCalculateAbcAction
        extends WindowCalculateUnification {
public class WindowCalculateFullCostAction
        extends WindowCalculateUnification {
public class WindowCalculateSebraeAction
        extends WindowCalculateUnification {
```

Figura 32 - Implementação de herança nas subclasses *WindowCalculateAbcAction, WindowCalculateFullCostAction* e *WindowCalculateSebraeAction*
Fonte: Autoria própria

Foi identificado que o módulo *app* segue um padrão de implementação, havendo uma implementação de menu, busca, cálculo, adição de atributos e alimentação do sistema para cada um dos 3 métodos de cálculo de preço de venda, sendo assim, há muitos métodos dentro das classes do módulo que se repetem. Essa repetição de métodos é igual ao problema que já foi apresentado e solucionado com a utilização do metapadrão *Unification*. Para cada conjunto de classes onde os métodos se repetem foi realizado o mesmo procedimento da

aplicação do metapadrão *Unification*, assim não serão apresentados em detalhes a implementação em todos os conjuntos de classes.

Dentro do módulo *app* não foram identificados outras partes de código que pudessem ser aplicados os outros metapadrões, portanto dentro do módulo *app* o único metapadrão aplicado foi o *Unification*.

No módulo *app* foi identificado que algumas classes continham métodos longos e eles faziam mais do que a sua real finalidade. Um desses métodos é o *save* presente dentro da classe *WindowAddFullCostAction*, que contém uma verificação do preenchimento de todos os campos obrigatórios e caso tudo estivesse correto ele apresentava para o usuário que o registro foi cadastrado ou editado com sucesso. Essas operações não são de responsabilidade do método *save*, deixando o método sobrecarregado.

Para solucionar o problema do método *save* foi utilizada a técnica de refatoração *Extrair Método* definida por Fowler (1999), que consiste em extrair o código não pertinente ao método e criar um novo método com este código. A figura 33 mostra o trecho de código não pertinente ao método *save*.

```
if ("cadastrado com sucesso".equals(result)) {
    ActionMessages messages = new ActionMessages();
    messages.add("messages.detail", new ActionMessage("message.save.success"));
    saveMessages(request, messages);
} else if (result.equals("editado com sucesso")) {
    ActionMessages messages = new ActionMessages();
    messages.add("messages.detail", new ActionMessage("message.edit.success"));
    saveMessages(request, messages);

} else if (result.equals("campos obrigatorios faltando")) {
    ActionErrors erros = new ActionErrors();
    erros.add("errors.detail", new ActionMessage("alert.fields.required"));
    saveErrors(request, erros);
}
```

Figura 33 - Trecho de código do método *save*
Fonte: Autoria própria

Para solucionar o problema apresentado foi criada uma nova classe chamada *WindowResultPresentation* e dentro desta classe foi criado método *show* que contém o código retirado do método *save*. Foi decidido pela criação de uma nova classe e extrair o método para esta classe pois o método *save* com o código com problema está presente em outras 4 (quatro) classes, logo para estas também é possível utilizar a mesma solução, evitando assim a duplicação de código, que ocorreria se fosse extraído o código do método *save* para cada

classe onde ele é implementado. A criação classe *WindowResultPresentation* com o método *show* é mostrado na figura 34.

```
public class WindowResultPresentation extends LookupDispatchAction {

    public void show(String result, HttpServletRequest request){
        if ("cadastrado com sucesso".equals(result)) {
            ActionMessages messages = new ActionMessages();
            messages.add("messages.detail", new ActionMessage("message.save.success"));
            saveMessages(request, messages);
        } else if ("editado com sucesso".equals(result)) {
            ActionMessages messages = new ActionMessages();
            messages.add("messages.detail", new ActionMessage("message.edit.success"));
            saveMessages(request, messages);
        } else if ("campos obrigatorios faltando".equals(result)) {
            ActionErrors erros = new ActionErrors();
            erros.add("errors.detail", new ActionMessage("alert.fields.required"));
            saveErrors(request, erros);
        }
    }

    @Override
    protected Map<String, String> getKeyMethodMap() {
        return null;
    }

}
```

Figura 34 - Classe *WindowResultPresentation*
Fonte: Autoria própria

O código dentro do método *save* também foi modificado, agora é criado um objeto da classe *WindowResultPresentation* e esse objeto faz a chamada para o método *show* que faz o trabalho de verificar e apresentar para o usuário o resultado. O novo código do método *save* é mostrado na figura 35.

```
WindowResultPresentation confirmation = new WindowResultPresentation();
confirmation.show(result, request);
```

Figura 35 - Novo código do método save
Fonte: Autoria própria

Após a aplicação do passo das técnicas de refatoração no FrameMK, volta-se para o passo de aplicar a ferramenta para ver se ainda existem outras refatorações possíveis no módulo. Não havendo novas refatorações no módulo, segue-se para a validação do sistema, onde foi verificado se as funcionalidades do FrameMK continuam corretas após o processo de refatoração.

Para fazer a validação necessária, foi feito as mesmas interações realizadas no passo 1 da etapa 1 de entender o sistema, verificando se estão corretos as funções de alimentar o sistema e realizar o cálculo do preço de venda

de um produto ou serviço. Realizando a mesma sequência de passos utilizados na etapa 1, o sistema respondeu e funcionou corretamente, assim como estava funcionando antes da refatoração, sendo possível adicionar atributos e realizar o cálculo do preço de venda.

A próxima iteração do processo de refatoração é verificar se o próximo módulo da ordenação será refatorado, como apresentado anteriormente. Neste trabalho somente o módulo *app* foi refatorado, logo as próximas iterações até o final de todos os módulos serão falsas até chegar ao fim desta etapa.

5.2 ESTATÍSTICAS REFERENTE A APLICAÇÃO DO MÉTODO

Esta seção tem como objetivo apresentar alguns dados sobre a refatoração do módulo *app* do framework de domínio FrameMK a partir do método proposto neste trabalho.

Para a refatoração do FrameMK foram modificadas ao todo 30 (trinta) classes dentro do módulo *app*, 23 (vinte e três) classes do módulo *Persistence* e *BussinnesRule* e 5 (cinco) classes do módulo *service*.

Foram aplicadas 295 (duzentos e noventa e cinco) refatorações sugeridas pela ferramenta *SonarQube* (2008), e utilizando o cálculo de porcentagem, chega-se ao resultado de que foram retirados 61% dos *bad smells* presentes dentro do módulo *app*.

A partir da ferramenta *SonarQube* também foi possível medir o *SQALE rating* do módulo *app*, que tem como objetivo medir o quão objetivo, preciso, de fácil reprodução e automatizado é o código, sendo que antes da refatoração a nota do módulo *app* no *SQALE rating* era B e após a refatoração ela subiu para a nota A. A tabela 1 ilustra como eram algumas características do FrameMK antes e depois da refatoração.

Tabela 1 - Estatísticas da refatoração do FrameMK

	Antes	**Depois**
Quantidade de Classes	37	39
Quantidade de *bad smells*	477	182
Complexidade	531	236
Duplicações	39,2%	35,8%
Linhas de código	2797	2921
Quantidade de funções	144	151
SQALE rating	B	A

Fonte: Autoria própria

Utilizando a técnica de Fowler foi possível aplicar uma refatoração dentro do módulo *app*, que gerou a redução da duplicação de código de 3 (três) classes, *WindowCalculateFullCostAction*, *WindowCalculateAbcAction* e *WindowCalculateSebraeAction* onde continham códigos idênticos.

Foi utilizado também o metapadão *Unification*, onde foi necessário a criação de uma nova classe, sendo assim foi possível unificar 3 (três) métodos utilizados por 3 (três) classes diferentes.

5.3 ANÁLISE DO MÉTODO PROPOSTO

Durante o processo de refatoração foram identificados pontos onde se houve uma maior facilidade para a aplicação do método e pontos de maior dificuldade.

Na primeira etapa *Entender o sistema,* o framework FrameMK já era conhecido pelo autor, assim tinha-se um conhecimento prévio de como interagir com o framework, porém em alguns casos o desenvolvedor que irá aplicar a refatoração utilizando o método proposto pode não ter um conhecimento prévio sobre o framework que irá refatorar, o que pode levar um pouco mais de tempo para se executar a primeira etapa. A maior dificuldade da primeira etapa foi realizar a engenharia reversa do código para gerar o diagrama de classes, pois o FrameMK conter 344 classes, e que muitas delas não possuem dentro de suas estruturas os atributos de outras classes as quais estão relacionadas.

A segunda etapa em que os módulos são ordenados também não houve dificuldade, pois foi implementado o algoritmo descrito na etapa. Sendo assim, foi necessário apenas analisar o módulo a partir da ferramenta *SonarQube* e utilizar os dados obtidos sobre a quantidade de *bad smells* como entrada no algoritmo de ordenação. Este processo poderia ser integrado com a ferramenta, visto que os dados são obtidos por meio dela. Isto representa uma restrição que deve ser resolvida se o método proposto for automatizado.

A terceira etapa em que o sistema foi refatorado, houve maiores dificuldades em relação as outras, pois era necessário um bom conhecimento sobre as técnicas de refatoração, padrões de projeto, inversão de controle e metapadrões para que fosse possível identificar partes de códigos no framework onde era possível e necessário sua aplicação.

Em comparação com os outros métodos, o método proposto tenta unir o que há de melhor entre os métodos de Rapeli (2006) e Mens e Tourwé (2004) podem oferecer no contexto de frameworks. Assim, como o método de Rapeli (2006), o método proposto também oferece uma forma melhor de se entender como o sistema funciona, contendo a etapa de gerar diagrama de classes. Também oferece a aplicação de padrões de projeto de forma específica, como um meio de se refatorar o framework.

Em comparação com o método de Mens e Tourwé (2004), o método proposto também oferece a funcionalidade de poder ser aplicado a qualquer linguagem de programação, diferente do método de Rapeli (2006) que é aplicado apenas em sistemas Java. No método proposto também é possível identificar os tipos de refatoração a serem aplicados, assim como o método de Mens e Touwé (2004).

O Quadro 12 apresenta a comparação entre o método de Rapeli (2006), Mens e Tourwé (2004) e o método proposto neste trabalho.

Características	Rapeli (2006)	Mens e Tourwé (2004)	Método Proposto
Compreender a funcionalidade do sistema.	X	X	X
Gerar diagrama de classe.	X		X
Aplicar padrões de projeto de forma específica.	X		X
Testar sistema refatorado.	X	X	
Refatorar para qualquer linguagem.		X	
Analisar código para refatorar.	X	X	X
Identificar tipo de refatoração a serem aplicadas.		X	X
Utilizar ferramentas automatizadas para análise do código.			X
Aplicar Inversão de Controle			X
Aplicar Metapadrões			X

Quadro 12 - Quadro de comparação entre os métodos de Rapeli (2006), Mens e Tourwé (2004) e Método Proposto
Fonte: Autoria própria

Diferente dos métodos de Rapeli (2006) e Mens e Tourwé (2004), o novo método apresenta a utilização de ferramentas automatizadas, que auxiliam na refatoração do framework, identificando mais facilmente pontos do código que necessitam de atenção. Também apresenta a refatoração a partir da utilização de Inversão de Controle e Metapadrões, que são dois conceitos utilizados na criação de um framework, logo devem ser levados em consideração na refatoração.

O método proposto não contempla etapas que devem ser realizadas para a fase de teste, somente explicita que a validação deve ser realizada, mas não demonstra como. Por isto no quadro 12, na característica *Testar sistema refatorado* foi deixada em branco.

6 CONCLUSÃO

Este trabalho apresentou um novo método para a refatoração, porém diferente dos métodos já publicados, pois foca no contexto de *frameworks* de domínio em que se utilizou as características tais como: padrões de projeto, metapadrões e inversão de controle, além de utilizar também técnicas de refatoração.

O método proposto é composto por 3 (três) etapas principais: *Entender o sistema*, *Ordenar módulos* e *Refatorar módulo*.

A primeira etapa *Entender o sistema* tem como objetivo permitir ao desenvolvedor conhecer o sistema em que irá trabalhar, seguindo os passos de *Utilizar o sistema*, para poder se ambientar, *Verificar módulos*, onde é identificado os módulos que compõe o sistema e *Gerar diagrama de classe dos módulos*, em que se cria o diagrama de classe dos módulos para poder verificar de uma forma mais fácil como os módulos se relacionam. Os passos *Utilizar o sistema* e *Gerar diagrama de classe* foram concebidos a partir do método de Rapeli (2006).

A segunda etapa *Ordenar módulos* apresenta um algoritmo de ordenação para que os módulos contemplados em um framework possam ser classificados em ordem crescente pela quantidade de *bad smells*.

A terceira etapa *Refatorar módulo* realiza a refatoração dos módulos efetivamente. Ela oferece a possibilidade do desenvolvedor escolher qual módulo deseja refatorar, ou seja, não o obrigando a refatorar todos os módulos. Nesta etapa o desenvolvedor aplica, se necessário, padrões de projeto, metapadrões, inversão de controle e corrige os *bad smells* que foram detectados pela ferramenta de refatoração.

O guia de catalogação da aplicação de metapadrões e inversão de controle foi realizado também na terceira etapa. Estes foram construídos a partir de exemplos práticos da aplicação desses conceitos, baseado no trabalho de Gura e Carmo (2009), e depois criou-se os modelos genéricos.

O método proposto foi aplicado no estudo de caso FrameMK, que é desenvolvido e mantido pelo Grupo de Pesquisa em Sistemas de Informação (GPSI). A aplicação contemplou todas as etapas e passos propostos, desde o

entendimento do sistema, seguindo para a ordenação dos módulos até chegar na etapa de refatoração, onde foi aplicada as características identificadas como importantes para um framework de domínio.

Com as refatorações efetuadas no FrameMK conseguiu-se diminuir a quantidade de *bad smells*, aumentar a flexibilidade, diminuir a complexidade, reduzir a duplicação de código, entre outros.

A diferença do método proposto em relação aos métodos da literatura são: uso de ferramentas de análise de código, criação do guia para a aplicação de metapadrões e inversão de controle. Em relação ao método de Mens e Tourwé (2004) e Rapeli (2006) possui as seguintes semelhanças: compreender o sistema e analisar código. Comparando com Mens e Touwé (2004) o método possui formas de identificar onde a refatoração será aplicada. Considerando o método de Rapeli (2006), a igualdade está nas tarefas de aplicar padrões de projeto e criar o diagrama de classe.

6.1 TRABALHOS FUTUROS

Os trabalhos futuros que podem ser realizados a partir desta pesquisa são:

- A identificação de novas características na construção de frameworks de domínio que devem ser levados em consideração no processo de refatoração.
- A aplicação de métricas para avaliar quantitativamente a contribuição do método na refatoração de frameworks.
- Aplicar o método proposto em outros estudos de caso ou módulos.
- Automatizar o processo de detecção de metapadrões em código fonte.
- Refinar o modelo de classes do FrameMK de modo a facilitar a engenharia reversa.
- Definir as etapas que devem ser realizadas na fase de validação do framework.

REFERÊNCIAS

ANDRADE, V. C.; CAPELLER, P. E. B.; MATOS, S. N. Identificação dos pontos de estabilidade e flexibilidade no modelo de requisitos dos métodos de preço de venda. **Anais** SULCOMP, v. 0, n. 0, set. 2010.

Apache Tomcat. Disponível em: < http://www. http://tomcat.apache.org/>. Acesso em: 2 set. 2015.

BARROS, Víctor. Escolha da ferramenta para a refatoração no FrameMK. Disponível em: <http://gpes.pg.utfpr.edu.br/gpes/arquivos/material/engenharia/refatoracao/2014/RS.BARROS.2014.L.pdf>. Acesso em: 10 out. 2015

BEN-ABDALLAH, Hanêne; BOUASSIDA, Nadia; GARGOURI, Faiez; BEM-HAMADOU, Abdelmajid. A UML-Based Framework Design Method. **Journal of Object Technology**, V.3, N.8, pág.98-119, 2004.

BRAGA, Rosana Terezinha Vaccare. **Um processo para Construção e Instanciação de Frameworks Baseados em uma Linguagem de Padrões para um Domínio Específico**. Tese (Doutorado em Ciência da Computação) – Universidade de São Paulo, São Carlos, 2003.

BUTLER, Greg; CHEN, Ling; CHEN, Xuede; GAFFAR, Ashraf;LI, Jinmiao; XU, Li Lugang. The Know-It-All-Project: A Case Study in Framework Development and Evolution. **In**: Domain Oriented Systems Development: Perspectives and Practices, Proceedings Taylor & Francis Publishers. v.1, p.101-117. London. 2002.

CAMARGO, Valter Vieira; MASIERO, Paulo César. Uma Abordagem de Evolução de Sistemas Orientados a Objetos Apoiada por Frameworks Transversais. **Anais**...SBES 05. Uberlândia, pág. 200-215, 2005.

Checkstyle 6.7. Disponível em: <http://checkstyle.sourceforge.net/>. Acesso em: 2 jun. 2015.

Codepro Analytix. Disponível em: <https://developers.google.com/java-dev-tools/codepro/>. Acesso em: 10 jun. 2015.

Eclipse. Disponível em: <http://www.eclipse.org/>. Acesso em: 10 jun. 2015.

Enterprise Architect. Disponível em: <http://www.sparxsystems.com.au/>. Acesso em: 23 set. 2015.

FAYAD, Mohamed E.; SCHMIDT, Douglas C. Object-oriented application frameworks. **Magazine Communications of the ACM**, v.40, n.10, p.32-38, 1997.

FAYAD, Mohamed E.; SCHMIDT, Douglas C.; JOHNSON, Ralph E. **Building Application Frameworks: Object Oriented Foundations of Framework Designs**. John Wiley & Sons, Inc., 1999.

FENTON, Norman E.; PFLEEGER, Shari Lawrence. **Software Metrics: A Rigorous and Practical Ap-proach**. 2 edition ed. [S.l.]: Course Technology, 1998. 656 p.

FindBugs 3.0.1. Disponível em: <http://findbugs.sourceforge.net/>. Acesso em: 2 jun. 2015.

Firebird 2.5. Disponível em: < http://www.firebirdsql.org/>. Acesso em: 2 set. 2015.

FOWLER, Martin. **Refactoring:** Improving the Design of Existing Code. Boston: Addison-wesley Professional, 1999. 464 p.
FrameMK. Disponível em: <http://gpes.pg.utfpr.edu.br/framemk/>. Acesso em: 10 ago. 2015.

GAMMA, E.R.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns: Elements of Reusable Object-Oriented Software.** 1 ed, Estados Unidos: Addison-Wesley, 1994.

GPSI. Grupo de Pesquisa em Sistemas de Informação: GPSI. Disponível em: <http://gpes.pg.utfpr.edu.br/gpes/>. Acesso em: 27 abr. 2015.

GURA, Evandro R.; CARMO, Ewerson R.; **Contextualização de Metapadrões no Desenvolvimento de Aplicações Desktop**. 2009. 87 f. Trabalho de Conclusão de Curso – Tecnologia Análise e Desenvolvimento de
In:8 Conference on Oriented-Programming: Systems, Languages and Applications, **Proceedings…** of The OOPSLA – Tutorial Notes, p. 567-617, 1993.

JOHNSON, Ralph. How To Design Frameworks. In:8 Conference on Oriented-Programming: Systems, Languages and Applications, **Proceedings…** of The OOPSLA – Tutorial Notes, p. 567-617, 1993.

LANDIN, Niklas; NIKLASSON, Axel. **Development of Object-Oriented Frameworks**. Master Thesis (Doctoral in Communication Systems – Lund, Sweden, 1995.

LEHMAN, Meir M. Programs, Life Cycles, and Laws of Software Evolution. In: **Proceedings** of the IEEE, p. 1060-1076, 1980.

MATOS, Simone Nasser. **Um Método Dirigido por Responsabilidades para Obtenção Antecipada de Pontos de Estabilidade e de Flexibilidade no Desenvolvimento de Frameworks de Domínio**. Instituto Tecnológico de Aeronáutica. Divisão de Ciência da Computação. São José dos Campos, 2008.

MATOS, Simone Nasser; FERNANDES, Clovis Torres. **Um Panorama dos Processos de Desenvolvimento de Frameworks de Domínio**. Instituto Tecnológico de Aeronáutica. Divisão de Ciência da Computação. CTA/ITA-JEC/RP-001/2007.São José dos Campos, 2007.

MATTSON, Michael. **Object-Oriented Frameworks: A Survey of Methodological Issues**. Licentiate Thesis, Department of Computer Science, Lund University, CODEN: LUTEDX/(TECS-3066)/1-130/(1996), also as Technical Report, LU-CS-TR: 96-167, Department of Computer Science, Lund University, 1996.

MAZER JUNIOR, A. **Métodos de Formação de Preço de Venda em Sistemas ERP por Intermédio de Arquitetura Orientada à Serviços do Framework FrameMK.** 2013. 115 f. Dissertação (Mestrado em Engenharia de Produção) – Programa de Pós-Graduação em Engenharia de Produção, Universidade Tecnológica Federal do Paraná, Ponta Grossa, 2013.

MENS, Tom. TOURWÉ, Tom. A Survey of Software Refactoring. **IEEE Transactions on Software Engineering**, v. 30, n. 2, 2004.

MILLS, Everald E. **Software Metrics**. Software Engineering Institute/SEI - Carnegie Mellon University, 1998.

NetBeans IDE 8.0.2. Disponível em: <https://netbeans.org/community/releases/80/>. Acesso em: 10 jun. 2015.

OPDYKE, William F. **Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks**. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

PREE, Wolfgang. **Design Patterns for Object-Oriented Software Development**. Addison-Wesley Publishing Company, ACM Press, 1995.

PREE, Wolfgang. Rearchitecturing Legacy Systems: Concept & Case Study. In:8 First Working IFIP Conference on Software Architecture, **Proceedings…** of WICSA'99 – San Antonio, Texas, 22-24 February 1999.

PRESSMAN, Roger S. **Engenharia de Software:** Uma Abordagem Profissional. McGraw Hill. Nova Iorque, 2010.

RAPELI, Leide R. **Refatoração de sistemas Java utilizando padrões de projeto: um estudo de caso**. 2006. 127 f. Dissertação (Mestrado, Universidade Federal de São Carlos. São Paulo, 2006).

RIBAS, J. H. **Desenvolvimento de Classes de Teste para a Camada de Persistência do Framework de Preço de Venda (FrameMK) usando JUnit.** 2014. Trabalho de Conclusão de Curso (Graduação) – Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas. Universidade Tecnológica Federal do Paraná, Ponta Grossa, 2014.

SILVA, Ricardo Pereira. **Suporte ao Desenvolvimento e ao Uso de Frameworks e Componentes**. Dissertação de Mestrado. Instituto de Informática. Programa de Pós-Graduação em Computação. Universidade Federal do Rio Grande do Sul, 2000.

Sistemas, Universidade Tecnológica Federal do Paraná, Ponta Grossa, 2009.

SOMMERVILLE, Ian. **Engenharia de Software**. Addison Wesley: São Paulo, 2011.

SONARQUBE 5.1.1. Disponível em: <http://www.sonarqube.org/>. Acesso em 2 jun. 2015.

SQALE rating. Diponível em: <http://www.sonarsource.com/products/plugins/governance/sqale/>

TALIGENT. **Building Object-Oriented Frameworks**. A Taligent White Paper. 1994.

WILSON, David, A.; WILSON, Stephen, D. Writing Frameworks – Capturing Your Expertise About a Problem Domain. In: 8 Conference on Object-Oriented Programming: Systems, Languages and Aplications. **Proceddings …** OOPSLA – Tutorial Notes, 1993.

ANEXO A – QUADRO DE REFATORAÇÕES BASEADO EM PADRÕES DE PROJETO

O Quadro 13 foi retirada na integralmente do trabalho de Rapeli (2006).

<table>
<tr><th>Padrão</th><th>Indícios</th></tr>
<tr><td colspan="2">Categoria: Criação</td></tr>
<tr><td>Abstract Factory</td><td rowspan="2">O código deve apresentar um ponto em que há a possibilidade de instanciação de uma ou mais classes. As possíveis classes a serem instanciadas devem ser subclasses da mesma superclasse, implementar a mesma interface ou serem dependentes umas das outras. Exemplo:
<pre>if (condicao1){
      ClasseA classeA = new ClasseA();
       ClasseB classeB = new ClasseB();
}else if (condicao2){
      ClasseC classeC = new ClasseC();
}
ClasseD classeD = new ClasseD();
ClasseE classeE = new ClasseE();</pre>
Utilizar o padrão Abstract Factory quando for necessária a delegação para a instanciação de objeto(s) e o padrão Factory Method quando for necessária a herança para a instanciação de objeto(s).</td></tr>
<tr><td>Factory Method</td></tr>
<tr><td>Builder</td><td>O código deve criar vários objetos (partes) para a criação de um objeto mais complexo, sendo que o processo de criação de todos os objetos é o mesmo. Essas partes devem ser escolhidas de acordo com dados pré-existentes. Exemplo:
<pre>if (condicao1){
   button.setVisible(true);
   label.setText(“objeto 1”);
}else if (condicao2){
   jComboBox1.setVisible(true);
   jComboBox1.addItem(“obj1”);
   jComboBox1.addItem(“obj2”);
}</pre></td></tr>
</table>

<table>
<tr><td>Prototype</td><td>O código deve apresentar a criação de dois ou mais objetos do mesmo tipo e a cópia de valores de atributos de um objeto para os respectivos atributos de outro objeto.
Exemplo 1:
<pre>Objeto objetoA = new Objeto();
Objeto objetoB = new Objeto();
objetoB.setCodigo(objetoA.getCodigo());
objetoB.setNome(objetoA.getNome());
objetoB.setCampos(objetoA.getCampos());</pre>Exemplo 2:
<pre>JTextField jTextA = new JTextField();
JTextField jTextB = new JTextField();
jTextB.setText(jTextA.getText());</pre></td></tr>
<tr><td>Singleton</td><td>O código deve apresentar trechos de controle de um recurso, que deve ser utilizado por um cliente de cada vez. Exemplo:
<pre>ConexaoBancoDados conexao = new ConexaoBancoDados();
if(banco não está ativo){
   conexao.conectar();
   //declarações utilizando o banco de dados
   conexao.desconectar();
}else if(banco está ativo){
   Sysout.println("Não foi possível conectar ao banco de
                                   dados: banco em atividade");
}</pre></td></tr>
<tr><td colspan="2">Categoria: Estrutural</td></tr>
<tr><td>Adapter</td><td>O código deve apresentar classes clientes (por exemplo, classe A) que acessam outras (por exemplo, classe B), sendo que não é toda a funcionalidade necessária à classe A que é provida pelo método utilizado, da classe B. O código deve mostrar adaptações realizadas por A sobre os resultados fornecidos por B. Exemplo:
<pre>class A{//classe cliente
...
  private B b = new
B(params);
...
  forma = b.Desenhar(params);
  forma.colorir();
  forma.mostrarDesenho();
...
}</pre><pre>class B{
...
  public Desenho
Desenhar(params){
    //desenha
  }
...
}</pre></td></tr>
</table>

Bridge	O código deve apresentar o mesmo dado de várias formas. Entretanto, o mesmo modelo de dados deve ser utilizado para as distintas apresentações. Exemplo: a apresentação dos dados de uma tabela pode ser feita mostrando-se os dados na própria tabela, em uma lista, em um gráfico, etc.
	`//produtos é um vetor com os dados a serem apresentados`
	`JList lista = new JList();`
	`JTable tabela = new Tabela();`
	`for(int i = 0; i < produtos.size(); i++){`
	`  lista.add(produtos.get(i));//adiciona produtos na lista`
	`  //adiciona produtos na tabela`
	`}`
	`//exibe lista`
	`//exibe tabela`

Composite	O código deve apresentar hierarquização de algum objeto, de modo que possa ser mapeado (ou já esteja) em uma estrutura de árvore. O tratamento para objetos da mesma hierarquia deve ser igual. No código podem ser encontrados objetos das bibliotecas `DefaultMutableTreeNode`, `Vector` e/ou `Enumeration`.

Decorator	O código deve apresentar agregação de funcionalidade(s) a um ou mais objetos, em tempo de execução. Exemplo:
	`if(cliente é homem)`
	`  cliente.AdicionaClienteSorteio();`

Façade	O código deve apresentar várias classes clientes que acessam os mesmos recursos, ou seja, métodos de outras classes. Esses recursos podem estar espalhados pelo sistema e são os candidatos a comporem a interface *Façade*, sendo que há a necessidade de simplificar as interfaces entre as classes.

<table>
<tr>
<td>Flyweight</td>
<td>O código deve apresentar a instanciação de muitos objetos da mesma classe, que sejam diferenciados apenas por poucos parâmetros. Exemplo:
<pre>
private <<tipo>> obj1 = new <<tipo>><<params>>;
private <<tipo>> obj2 = new <<tipo>><<params>>;
private <<tipo>> obj3 = new <<tipo>><<params>>;
private <<tipo>> obj4 = new <<tipo>><<params>>;
...
private <<tipo>> objN = new <<tipo>><<params>>;

obj1.setAtt1(valor);   obj2.setAtt1(valor);
obj3.setAtt1(valor);   obj4.setAtt1(valor);
objN.setAtt1(valor);

obj1.setAtt3(valor1); obj2.setAtt3(valor1);
obj3.setAtt3(valor1); obj4.setAtt3(valor1);
objN.setAtt3(valor1);
...
obj1.setAtt2(valorA); obj2.setAtt2(valorB);
obj3.setAtt2(valorC); obj4.setAtt2(valorD);
objN.setAtt2(valorN);
</pre></td>
</tr>
<tr>
<td>Proxy</td>
<td>O código deve apresentar alguma ação sendo executada enquanto o objeto que já existe (virtual proxy) é utilizado/apresentado.
Exemplo 1:
<pre>
while(banco banco de dados está sendo inicializado)
   System.out.print(“Aguarde, iniciando banco de dados”);
</pre>
Exemplo 2:
<pre>
if(ícone foi criado)
   //desenha ícone na tela
else {
   //apresenta uma mensagem contendo o status do
   //carregamento da imagem
}
</pre></td>
</tr>
</table>

<table>
<tr><th colspan="2">Categoria: Comportamental</th></tr>
<tr><td>Chain of Responsability</td><td>O código deve permitir que várias classes tratem de uma requisição sem que uma tenha conhecimento da capacidade de tratamento da outra e sem que o cliente saiba qual classe deve tratá-la. A requisição é transferida entre as classes, até que uma possa atendê-la. Pode ou não haver prioridade entre as classes na transferência de uma requisição, de modo que, para que uma classe possa tratá-la, seja necessário que outra a trate antes. Exemplo:
<pre>if(condicao1){ //sem prioridade de tratamento
   classeA.Metodo1();
}else if(condicao2){ //com prioridade de tratamento
   classeB.Metodo2();
   classeC.Metodo3();
   classeA.Metodo1();
}</pre></td></tr>
<tr><td>Command</td><td>O código deve possuir ações a serem executadas quando houver interação do usuário com o sistema.
<pre>class A implements ActionListener{
...
 obj1.addActionListener(this);
 obj2.addActionListener(this);
 ...
 public void actionPerformed(ActionEvent event){
   if(event.getSource() == obj1)
      Metodo1();
   else if(event.getSource() == obj2)
      Metodo2();
 }
}</pre></td></tr>
<tr><td>Interpreter</td><td>O código deve permitir que suas operações sejam representadas como uma linguagem. Exemplo: analisador de expressões aritméticas.</td></tr>
<tr><td>Iterator</td><td>O código deve apresentar um objeto do tipo lista ou coleção que deve ser percorrido, de modo a identificar os elementos nele armazenados. Exemplo:
<pre>Vector v = new Vector();
...
for(int i = 0; i < v.size(); i++)
   Sysout.println(v.get(i));</pre></td></tr>
<tr><td>Mediator</td><td>O código deve apresentar interações complexas entre os componentes visuais do sistema, sendo que cada componente necessita da informação sobre um ou vários outros componentes, há forte acoplamento. Exemplo:
<pre>public <<tipo_ret>> Metodo1(<<params>>){
   componente1.setVisible(false);
   componente2.setEnabled(true); }
public <<tipo_ret>> Metodo2(<<params >>){
   componente1.setVisible(true);
   componente1.setText("texto texto");
   componente2.setEnable(false);
   componente3.setBackground(cor);
}</pre></td></tr>
</table>

Memento	O código deve apresentar o armazenamento do estado interno de um objeto e recuperação deste estado posteriormente. Exemplo: implementação de funções de desfazer/refazer.
	`//cliente é um objeto do tipo Cliente`
	`//vetor é um objeto do tipo Vector`
	`public void armazenarCliente(){`
	`  vetor.add(cliente.getCodigo());`
	`  vetor.add(cliente.getNome());`
	`  vetor.add(cliente.getRG());`
	`  ...`
	`}`
	`public void recuperarCliente(){`
	`  cliente.setCodigo(vetor.get(i++));`
	`  cliente.setNome(vetor.get(i++))`
	`  cliente.setRG(vetor.get(i++));`
	`  ...`
	`}`

Observer	O código deve apresentar dados de várias maneiras ao mesmo tempo, sendo que quando os dados de uma representação mudam os demais devem ser notificados da mudança. Exemplo:
	`...`
	`ObjetoA objA = new ObjetoA();`
	`ObjetoB objB = new ObjetoB();`
	`ObjetoC objC = new ObjetoC();`
	`...`
	`if(objA foi alterado OU objB foi alterado OU objC foi alterado){`
	`  alterarApresentacaoObjA();`
	`  alterarApresentacaoObjB();`
	`  alterarApresentacaoObjC();`
	`}`

State	O código deve comportar-se de maneira diferente de acordo com o estado do objeto. Deve haver declarações `if/else` ou `switch` extensas, que definem o comportamento a ser adotado. Exemplo:
	`if(componente.getColor() igual COR_A)`
	`  componente.setColor(corB);`
	`else if(componente.getColor() igual COR_B)`
	`  componente.setColor(corC);`
	`else if(componente.getColor() igual COR_C)`
	`  componente.setColor(corD);`

Strategy	O código deve requisitar uma funcionalidade particular, que pode ser fornecida por várias classes, com o mesmo nome e implementações distintas. Deve haver declarações `if/else` ou `switch`, definindo qual funcionalidade deve ser invocada. Exemplo:
	`if(condicao1)  objA.Metodo();`
	`else if(condicao2)  objB.Metodo();`
	`else  objC.Metodo();`

Template Method	O código deve apresentar uma classe com a possibilidade de um ou mais de seus métodos serem definidos em subclasses. Exemplo:

<table>
<tr>
<td></td>
<td>

```
public void login(){
   private JTextField usuario = new JTextField();
   private JPasswordField senha = new JPasswordField();
   private JButton btOk = new JButton("OK");
   //inserção dos componentes na tela

private void acaoBtOk(){
  if(usuario.equals(“antonio”))
     //inicializa sistema
  else
     System.out.print(“Acesso negado”);
}
```

Outra implementação possível para o método `acaoBtOK`:

```
private void acaoBtOk(){
  Conexao conexao = new Conexao();
  if(conexao.VerificarUsuario(usuario.getText()))
    //inicializa sistema
  else
    System.out.print(“Acesso negado”);
}
```

Neste exemplo, o método `acaoBtOk` pode ser definido em uma subclasse.
</td>
</tr>
<tr>
<td>Visitor</td>
<td>

O código deve realizar uma ou mais operações em objetos (de interfaces distintas) ou ainda, em um grande número de objetos. Exemplo: geração do objeto relatório, que deve recolher determinados dados de todos os objetos relacionados a ele.

```
//objetos é o vetor que contém todos os objetos
//a serem considerados na operação
for(int i = 0; i < objetos.size(); i++)
   soma = soma + objetos.getTotalGasto();
```
</td>
</tr>
</table>

Quadro 13 - Indícios no código para aplicação de padrões de projeto no sistema existente
Fonte: Rapeli (2006, p. 28)

APÊNDICE A – QUADRO COMPLETO DAS CATEGORIAS E TÉCNICAS DE REFATORAÇÃO RETIRADAS DO SÍTIO DE FOWLER

O quadro 14 foi gerado a partir das categorias e técnicas de refatoração de Fowler (1999) mantidas em seu sítio.

Categoria	Técnica
Associations	*Change Bidirectional Association to Unidirectional* *Change Reference to Value* *Change Unidirectional Association to Bidirectional* *Change Value to Reference* *Duplicate Observed Data* *Extract Class* *Inline Class* *Replace Delegation With Hierarchy* *Replace Delegation With Inheritance* *Replace Inheritance With Delegation* *Replace Method With Method Object*
Encapsulation	*Encapsulate Collection* *Encapsulate Downcast* *Encapsulate Field* *Hide Delegate* *HIde Method* *Preserve Whole Object* *Remove Setting Method* *Self Encapsulate Field*
Generic Types	*Replace Array with Object* *Replace Data Value with Object* *Replace Hash with Object* *Replace Magic Number with Symbolic Constant* *Replace Record with Data Class*
Interfaces	*Extract Interface* *Replace Constructor with Factory Method*
Class Extraction	*Extract Class* *Extract Interface* *Extract Module* *Extract Subclass* *Extract Superclass* *Introduce Parameter Object*
GOF Patterns	*Form Template Method* *Replace Type Code with State/Strategy*
Local Variables	*Extract Variable* *Inline Temp* *Remove Assignments to Parameters* *Replace Method with Method Object* *Replace Temp with Chain* *Replace Temp with Query* *Split Temporary Variable*
Vendor Libraries	*Introduce Foreign Method* *Introduce Gateway* *Introduce Local Extension*
Errors	*Replace Error Code with Exception* *Replace Exception with Test*
Type Codes	*Replace Type Code with Class* *Replace Type Code with Module Extention* *Replace Type Code with Polymorphism* *Replace Type Code with State/Strategy*

	Replace Type Code with Subclasses
Method Calls	*Add Parameter* *Encapsulate Downcast* *Hide Method* *Introduce Expression Builder* *Introduce Gateway* *Introduce Named Parameter* *Introduce Parameter Object* *Parametize Method* *Preserve Whole Object* *Remove Control Flag* *Remove Named Parameter* *Remove Parameter* *Remove Setting Method* *Rename Method* *Replace Constructor with Factory Method* *Replace Error Code with Exception* *Replace Exception with Test* *Replace Parameter with Explicit Methods* *Replace Parameter with Method* *Separate Query from Modifier*
Organizing Data	*Change Bidirectional Association to Unidirectional* *Change Reference to Value* *Change Unidirectional Association to Bidirectional* *Change Value to Reference* *Duplicate Observed Data* *Eagerly Initialized Attribute* *Encapsulate Collection* *Encapsulate Field* *Lazily Initialized Attribute* *Replace Array with Object* *Replace Data Value with Object* *Replace Magic Number with Symbolic Constant* *Replace Record with Data Class* *Replace Subclass with Fields* *Replace Type Code with Class* *Replace Type Code with State/Strategy* *Replace Type Code with Subclasses* *Self Encapsulate Field*
Inheritance	*Collapse Hierarchy* *Encapsulate Downcast* *Extract Interface* *Extract Module* *Extract Subclass* *Extract Superclass* *Form Template Method* *Introduce Null Object* *Pull Up Constructor Body* *Pull Up Field* *Pull Up Method* *Pull Down Field* *Pull Down Method* *Replace Abstract Superclass with Module* *Replace Conditional with Polymorphism* *Replace Delegation with Hierarchy* *Replace Delegation with Inheritance*

	Replace Inheritance with Delegation *Replace Subclass with Fields* *Replace Type Code with Subclasses*
Conditionals	*Consolidate Conditional Expression* *Consolidate Duplicate Conditional Fragments* *Decompose Conditional* *Introduce Assertion* *Introduce Null Object* *Recompose Conditional* *Remove Control Flag* *Replace Conditional with Polymorphism* *Replace Exception with Test* *Replace Nested Conditional with Guard Clauses*
Moving Features	*Exctract Class* *Extract Module* *Hide Delegate* *Inline Class* *Inline Module* *Introduce Foreign Method* *Introduce Local Extension* *Move Field* *Move Method* *Remove Middle Man*
Composing Methods	*Consoliate Conditional Expression* *Decompose Conditional* *Extract Method* *Extract Surrounding Method* *Extract Variable* *Form Template Method* *Inline Method* *Inline Temp* *Move Eval from Runtime to Parse Time* *Remove Assignments to Parameters* *Replace Loop with Collecion Closure Method* *Replace Method with Method Object* *Replace Temp with Query* *Split Temporary Variable* *Substitute Algorithm*
Defining Methods	*Dynamic Method Definition* *Introduce Class Annotation* *Isolate Dynamic Receptor* *Remove Unused Default Parameter* *Replace Dynamic Receptor with Dynamic Method Definition* *Replace Method with Method Object*

Quadro 14 - Tabela completa das categorias e técnicas de refatoração descritas por Fowler
Fonte: Autoria própria