USANDO MEDIÇÃO DE CÓDIGO FONTE PARA *REFACTORING*

GLAUCO DE FIGUEIREDO CARNEIRO

Salvador

Abril – 2003

UNIVERSIDADE SALVADOR

PROGRAMA DE PÓS-GRADUAÇÃO EM REDES DE COMPUTADORES

USANDO MEDIÇÃO DE CÓDIGO FONTE PARA *REFACTORING*

GLAUCO DE FIGUEIREDO CARNEIRO

Dissertação apresentada à Universidade Salvador, como parte das exigências do Curso de Mestrado Profissional em Redes de Computadores, área de concentração em Redes de Computadores, para obtenção do título de "Mestre".

Orientador

Prof. Dr. Manoel Gomes de Mendonça Neto

Salvador
Abril - 2003

USANDO MEDIÇÃO DE CÓDIGO FONTE PARA *REFACTORING*

GLAUCO DE FIGUEIREDO CARNEIRO

Dissertação apresentada à Universidade Salvador, como parte das exigências do Curso de Mestrado Profissional em Redes de Computadores, área de concentração em Redes de Computadores, para obtenção do título de "Mestre".

APROVADA COM DISTINÇÃO em 14 de abril de 2003.

Banca Examinadora:

Prof. Dr. José Carlos Maldonado (Universidade de São Paulo)	Membro
Prof. Dr. Augusto Loureiro da Costa (Universidade Salvador)	Membro
Prof. Dr. Manoel Gomes de Mendonça Neto (Universidade Salvador)	Orientador

Salvador
Abril – 2003

Agradecimentos

Todo lutador oriental tem um grande mestre que serve de exemplo, guia intelectual e espiritual. Este mestre geralmente preocupa-se em ensinar a pescar e não simplesmente a fornecer o peixe.

Sinto-me privilegiado por ter tido não somente um mestre, mas vários ao longo destes anos vividos, desde a infância até os dias de hoje.

Eis que agora me proponho a receber o título de Mestre. Com certeza terei que seguir sempre aprendendo na vida, pois aí está a beleza da nossa existência, seguir sempre aprendendo, ensinando, reaprendendo,… Nesta espiral segue a humanidade, cada vez mais decobrindo e redescobrindo, tendo consciência das suas limitações naturais e intrínsecas, mas sempre procurando superá-las, sem necessariamente subestimá-las.

Obrigado a todos aqueles que dedicaram parcela do seu tempo e sua atenção não somente no desenvolvimento deste trabalho, mas que me servem de referência:

Minha mãe, Lucília.
Minha esposa, Cristiane.
Prof. Padre Ugo do Colégio Antônio Vieira (*in memoriam*).
Aos meus padrinhos, Adauto e Zilda e primos Dauton e Luzia.
Meus familiares.
Meus amigos.
Meus alunos da Universidade Salvador.
Aos professores e funcionários da Universidade Salvador e do Nuperc.

Em especial ao meu orientador, Professor Manoel Mendonça, pela atenção dada em todos os momentos solicitados e pela forma natural que tem de estimular a sua equipe de pesquisa não somente a alcançar os resultados, mas a aprender com os mesmos.

Agradeço a Deus pela oportunidade de realizar este trabalho.

Glauco de Figueiredo Carneiro

Salvador, 31 de março de 2003.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Carneiro, G. F. **Usando Medição de Código Fonte Para *Refactoring*.** 2003. 129 f. Dissertação (Mestrado em Redes de Computadores), Universidade Salvador. Salvador – Bahia - Brasil.

Palavras-chave:*refactoring*, métricas de *software*, Meta Pergunta Métrica *(Goal Question Metric)*, engenharia de *software*.

Refactoring – melhorando a estrutura interna do *software* sem modificação no seu comportamento observável – é um mecanismo importante para se evitar a degradação da qualidade do *software*. Fundamental para tal finalidade é a identificação de trechos do código fonte que apresentam oportunidades de *refactoring* – comumente chamadas de *bad smells*.

Esta dissertação propõe uma abordagem para auxiliar na detecção de *bad smells* através de medição de código fonte. Como para resolver este problema são necessárias várias outras etapas ainda não implementadas, esta dissertação tem como objetivo estabelecer mecanismos que representem um primeiro passo no sentido de detectar *bad smells* através da medição de código fonte. Para tal finalidade é realizado um estudo que relaciona métricas, *refactorings* e *bad smells*. O estudo é dividido em duas partes. A primeira parte – *top-down* – é baseada na aplicação analítica do paradigma Meta Pergunta Métrica (MPM). A segunda parte – *bottom-up* – é um estudo empírico do relacionamento entre métricas conhecidas de código fonte, *refactorings* e *bad smells*.

O estudo baseado na aplicação analítica do paradigma Meta Pergunta Métrica (MPM) identificou tipos de métricas necessários para a identificação dos *bad smells* propostos por Fowler. O estudo mostra que 75% das métricas necessárias não estão disponíveis, sendo que deste universo, aproximadamente 78% são factíveis e implementáveis e o restante são métricas fortemente dependentes de análise cognitiva e de difícil implementação.

O estudo empírico analisou 47 métricas de código fonte ao longo de um estudo de caso – estas métricas foram obtidas a partir de um conjunto de métricas comercialmente disponíveis em ferramentas de medição de *software*. O estudo de caso mediu a variação destas métricas ao longo da seqüência de 77 *refactorings.* Foram utilizados dois coeficientes criados nesta dissertação especialmente para associar métricas e *refactorings* – Coeficiente de Associação entre Métrica e *Refactoring* (CAMR) e Coeficiente de Associação Forte entre Métrica e *Refactoring* (CAFMR) – e os resultados destas associações são apresentados para os *refactorings* executados durante o estudo de caso.

ABSTRACT

Carneiro, G. F. **On the Use of Source Code Measurement for Refactoring**. 2003. 129 p. Master Thesis, Salvador University. Salvador, Bahia, Brazil.

Keywords: refactoring, software metrics, Goal Question Metric, empirical study, software engineering.

Software refactoring - improving the internal structure of the software without changing its observable behavior - is an important action towards avoiding software quality decay. Key to this activity is the identification of portions of the source code that offers opportunities for refactoring - the so called "code bad smells".

This dissertation proposes an approach to help on the detection of code bad smells through source code measurement. To solve this problem, however, other stages still not implemented are necessary. This dissertation focuses on the first step towards a concrete method to detect code bad smells through source code measurement. It presents a study that relates metrics, refactorings, and bad smells. Our study is broken into two parts. The first - top-down - part is based on the analytical application of the Goal-Question-Metric (GQM) method. The second - bottom-up - part is an empirical study on the relationship between well-known source code metrics, refactorings and code bad smells.

The GQM study identified the type of metrics that are needed for each of the bad smells proposed by Fowler. The study shows that 75% of the needed metrics are not available. But, 78% of those can be implemented, while the remainder is strongly dependent on human cognitive analysis, therefore difficult if not impossible to implement.

The empirical study analyzed 47 source code metrics over a case study - these metrics make up a comprehensive set among those commercially available on software measurement tools. The case study measured the variation of these metrics over a sequence of 77 refactorings. The study used two customized association measures to relate metrics and refactorings - metric-refactoring association coefficient (MRAC) and metric- refactoring strong association coefficient (MRSAC) - and the results of those associations are presented for the refactorings executed during the case study.

1. INTRODUÇÃO

Um problema comum em desenvolvimento de *software* é a possibilidade de degradação contínua do *software* à medida que novas funcionalidades vão sendo acrescentadas sem que sejam tomadas as providências necessárias para sua adaptação à estrutura já existente.

Aplicações usando linguagens orientadas a objeto podem ser reestruturadas a partir de *refactorings* (Fowler, 2000a) (Opdyke, 1992). Operações de *refactoring* reorganizam a hierarquia de classes e redistribuem as variáveis de instância e métodos. De acordo com Fowler (2000a), o objetivo do *refactoring* é tornar o *software* mais fácil de ser compreendido e modificado, reestruturando implementações existentes para torná-las mais flexíveis, dinâmicas e reutilizáveis.

Nesta dissertação são estabelecidos relacionamentos entre alguns dos principais tipos de *refactoring*s propostos por Fowler (2000a) e métricas obtidas através do código fonte do *software*. O uso de métricas tem potencial para auxílio na execução do processo de *refactoring* e torna seus resultados analisáveis quantitativamente. Como resultado, tem-se um processo de *refactoring* mais previsível e menos dependente de heurísticas pessoais.

Crítico para o sucesso desta nossa abordagem é a seleção e aplicação efetiva de métricas baseadas nos objetivos de *refactoring* da equipe de desenvolvimento ou manutenção da aplicação.

Esta dissertação apresenta um estudo com o objetivo de introduzir métricas no uso de *refactorings*. Esta é uma etapa essencial para o estabelecimento futuro de uma metodologia para auxílio na detecção de oportunidades de *refactoring* usando métricas.

Os capítulos seguintes desta dissertação estão organizados da seguinte forma: no Capítulo 2 é apresentada uma revisão bibliográfica para mostrar o contexto no qual foi desenvolvido o tema desta dissertação; no Capítulo 3 é apresentada metodologia para tornar mensuráveis os indicadores de necessidade de *refactoring*; no Capítulo 4 são apresentados dois estudos de caso para estabelecer relacionamento entre *refactorings, bad smells* - termo usado por Fowler (2000a) para descrever oportunidade de *refactoring* - e métricas e no Capítulo 5 são apresentadas as conclusões do trabalho realizado.

2. REVISÃO BIBLIOGRÁFICA

Neste capítulo serão apresentados o conceito, tipos, classificação e etapas de *refactoring*, assim como conceitos associados a métricas com o objetivo de possibilitar a compreensão da metodologia proposta para a associação entre *bad smells*, *refactoring*s e métricas a serem aplicadas nos estudos de caso realizados.

2.1. DEFININDO *REFACTORINGS*

O conceito de *refactoring* foi originalmente definido por Opdyke (1992) como uma transformação de programa que preserva o seu comportamento e redefinido por Roberts (1999) como uma transformação do programa onde uma pré-condição particular fosse satisfeita.

Refactoring é um processo de modificação de *software* de tal forma que não seja alterado o seu comportamento externo, mas que sejam proporcionadas melhorias na sua estrutura interna (Fowler 2000a). É um tipo de reorganização que tem os seguintes objetivos: melhorar o projeto do *software*, tornar o código mais facilmente compreensível, auxiliar na resolução de possíveis problemas, tornar mais dinâmica a evolução do *software* e conseqüentemente reduzir os custos a ela associados.

Proposto e avaliado a partir de 1990 (Opdyke e Johnson, 1990) (Griswold ,1991) (Opdyke, 1992) (Griswold e Notkin,1993), *refactoring* tem proporcionado um desenvolvimento crescente de práticas de desenvolvimento de *software* (Fowler, 2000a). Como exemplo, pode-se citar que um dos princípios básicos do *Extreme Programming* (Beck, 1999) é a realização de *refactoring* de forma contínua como parte fundamental do processo de desenvolvimento de *software*.

2.2. CLASSIFICANDO *REFACTORINGS*

Dentre algumas das classificações de *refactoring*, foram consideradas nesta dissertação as classificações por objetivo e nível de abstração.

2.2.1. Classificando *refactorings* por objetivos

Fowler (2000a) argumentou que os *refactorings* poderiam ser classificados por objetivos conforme apresentado na Tabela 2.1. Será apresentada a seguir cada uma das seis classificações com suas respectivas motivações para alguns dos *refactorings* que a compõem. Esta é uma oportunidade para se conhecer o potencial dos *refactorings* propostos por Fowler. Este próprio autor denomina a classificação por objetivos abordada nesta seção como "catálogo de *refactorings*", ponderando que se trata de um catálogo em fase inicial com a possibilidade de acréscimo de novos componentes (Fowler, 2000a). O conhecimento desta classificação é importante para que seja possível o uso dos seus componentes de forma combinada para a correção de um dado *bad smell*, conforme será apresentado na Seção 3.1 desta dissertação.

A relação de *refactorings* com suas respectivas denominações em inglês relatadas por Fowler em (Fowler, 2000a) encontra-se no Apêndice 03.

Tabela 2.1 Classificação de *Refactorings*

Classificação de *Refactoring*	Relação de *Refactorings*
Composição dos Métodos	Migração de Trecho de Código para um Novo Método, Dissolução de Método, Dissolução de Variável Temporária, Substituir Variável Temporária por Consulta, Introdução de Variável Esclarecedora, Substituir Variável Temporária por Duas ou Mais, Uso de Variável Temporária, Substituição de Método por Objeto, Substituição de Algoritmo
Movendo funcionalidades entre objetos	Migração de Método para Outra Classe, Migração de Variável para Outra Classe, Migração de Trecho de Código para uma Nova Classe, Dissolução de Classe, Delegação Oculta, Remoção de Delegação, Introdução de Método Estrangeiro, Introdução de Extensão Local

Tabela 2.1 Classificação de Refactorings (Continuação)

Classificação de *Refactoring*	Relação de *Refactorings*
Organização de Dados	Encapsulamento de Informações, Criação de Objeto para Tratamento de Informações, Modificação para Objeto do Tipo Referência, Modificação para Objeto do Tipo Valor, Substituição de Arranjo para Objeto, Separação da Interface da Lógica do Negócio, Transformação de uma Associação Unidirecional para Bidirecional, Transformação de uma Associação Bidirecional para Unidirecional, Inclusão de uma Constante para Representar um Número, Transformação de uma Variável Pública em Privada com Mecanismos de Acesso, Encapsulamento de uma Coleção, Substituição de Registro por Classe de Dados, Substituição de Tipo de Código por Classe, Substituição de Tipo de Código por Subclasses, Substituição de Tipo de Código por Objeto do Tipo Estado/Estratégia, Substituição de Subclasses por Variável
Simplificando Expressões Condicionais	Decomposição de Condicional, Consolidação de Expressão Condicional, Consolidação de Trechos Duplicados em Expressão Condicional, Remoção de Variável de Controle, Substituição de Condicionais Encadeadas por Cláusulas, Substituição de Condicional por Polimorfismo, Uso de Objeto Nulo, Introdução de Afirmação
Tornando a chamada de métodos mais simples	Renomeando Métodos, Adição de Parâmetros Remoção de Parâmetros, Separação de Consulta por Modificador, Parametrização de Método , Substituição de Parâmetros por Métodos, Uso de Objeto para Obtenção de Informações, Substituição de Parâmetro por Método, Substituição de Parâmetro por Objeto, Remoção de Método de Atribuição de Valores, Ocultação de Método em Relação a Outras Classes, Substituição de Construtor por Subclasses, Encapsulando Projeção de Tipo, Substituição de Código de Erro por Exceção, Substituição de Exceção por Teste

Tabela 2.1 Classificação de Refactorings (Continuação)

Classificação de Refactoring	Relação de Refactorings
Trabalhando com Generalizações	Migração de Variável para Superclasse, Migração de Método para Superclasse, Migração de Construtor para Superclasse, Migração do Método para Subclasse, Migração de Variável para Subclasse, Migração de Trecho de Código para uma Nova Subclasse, Migração de Trecho de Código para uma Nova Superclasse, Migração de Trecho de Código para uma Nova Interface, Junção de Superclasse e Subclasse, Migração de Trecho de Código para um Novo Método Modelo, Substituição de Herança por Delegação, Substituição de Delegação por Herança

a) Composição dos Métodos: conjunto de *refactorings* com o objetivo de posicionar o código dos métodos de forma apropriada. Freqüentemente ocorre o uso de *refactorings* associados à composição de métodos para que o comportamento dos objetos seja expresso de forma apropriada. Métodos de tamanho excessivo não são recomendados, pois contêm grande número de informações, geralmente acompanhados de lógicas complexas. O *refactoring Migração de Trecho de Código para um Novo Método* retira uma parte do código de um método já existente e leva-o para um novo método criado exclusivamente para tal fim. *Dissolução de Método* é exatamente o inverso: antes de remover o método, o seu código é migrado para um outro método. O maior problema no uso do *refactoring Migração de Trecho de Código para um Novo Método* ocorre com as variáveis locais, sendo as variáveis temporárias as maiores responsáveis por isto. Assim, quando tal caso ocorrer, deverá ser usado o *refactoring Substituir Variável Temporaria por Consulta* para que sejam removidas as variáveis temporárias. Caso a variável temporária seja usada para várias finalidades, deve-se usar *Substituir Variável Temporária por Duas ou Mais* para facilitar a substituição das mesmas. Em alguns momentos, pode ser complexo remover uma variável temporária, podendo ser aplicado o *refactoring Substituição de Método por Objeto*, possibilitando dividir um método com o mesmo custo de adicionar uma nova classe para tal finalidade. Parâmetros não representam problemas desde que não sejam usados no âmbito do método para atribuição de valores. Neste caso, deve-se aplicar *Uso de Variável Temporária*. Uma vez que o método tenha sido dividido, pode-se compreendê-lo bem

melhor. O algoritmo também pode ser melhorado para que se torne mais claro através do *refactoring Substituição de Algoritmo* (Fowler, 2000a).

b) Movendo funcionalidades entre objetos: escolher de forma apropriada o posicionamento dos membros (métodos e atributos) é de grande importância no projeto de objetos. Para se encontrar uma nova posição para as funcionalidades pode-se usar *Migração de Método para Outra Classe* e *Migração de Variável para Outra Classe*. Quando usados juntos, pode-se primeiro usar *Migração de Variável para Outra Classe* e depois *Migração de Método para Outra Classe*. Quando uma classe estiver com muitos métodos, pode-se usar *Migração de Trecho de Código para uma Nova Classe* para conduzir parte destes para uma nova classe. Caso uma classe esteja com pouca ou praticamente nenhuma responsabilidade, deve-se usar *Dissolução de Classe* para mover seus membros para outra classe já existente. Se outra classe está sendo utilizada, é aconselhável ocultar este fato através do *refactoring Delegação Oculta*. Às vezes, ocultando a classe delegada resulta em modificações na interface do proprietário, sendo necessário a aplicação do *refactoring Remoção de Delegação*. Os dois últimos *refactorings* deste grupo são *Introdução de Método Estrangeiro* e *Introdução de Extensão Local*. Eles são usados sempre que não for possível o acesso ao código fonte da classe, ainda que seja necessário mover-se responsabilidades para tal classe. Caso seja apenas um ou dois métodos, deve-se aplicar *Introdução de Método Estrangeiro*, para mais de dois métodos deve-se aplicar *Introdução de Extensão Local* (Fowler, 2000a).

c) Organização de Dados: neste grupo estão *refactorings* que contribuem para melhorar o tratamento de dados. Muitas vezes é necessária a criação de um dispositivo para possibilitar acesso aos dados armazenados por um objeto, para tal fim deve-se usar o *refactoring Encapsulamento de Informações*. A possibilidade de se definir novos tipos de dados amplia o que poderia ser feito somente com os tipos de dados de linguagens tradicionais. Pode-se iniciar com um valor de dados simples e depois verificar que um objeto pode ser mais adequado. *Criação de Objeto para Tratamento de Informações* permite que dados sejam convertidos em objetos. Quando tais objetos forem instâncias necessárias em várias partes de um programa deve-se aplicar *Modificação para Objeto do Tipo Referência*. Um vetor pode ser substituído através de *Substituição de Arranjo para Objeto*. Em todos estes casos, o objeto é o primeiro passo. O segundo passo é o uso de *Migração de Método para Outra Classe* para se adicionar comportamento aos

novos objetos. Números com significados especiais representam um problema. Para substituí-los use *Inclusão de uma Constante para Representar um Número*. Relacionamentos entre objetos podem ser uni e bidirecionais, assim, a depender do caso, pode-se usar *Transformação de uma Associação Unidirecional para Bidirecional* ou *Transformação de uma Associação Bidirecional para Unidirecional* para remover complexidade desnecessária quando não for necessária comunicação bidirecional. São comuns os casos em que classes de interfaces gráficas de usuário também ofereçam funcionalidades associadas à lógica do negócio. Para remover este comportamento para uma classe apropriada, deve-se usar *Separação da Interface da Lógica do Negócio*. Em relação ao uso do encapsulamento, quando dados estiverem com acesso público pode-se usar *Transformação de uma Variável Pública em Privada com Mecanismos de Acesso*. Caso seja uma coleção, deve-se usar *Encapsulamento de uma Coleção*. Caso um registro inteiro esteja acessível, deve-se usar *Substituição de Registro por Classe de Dados*. Uma forma de dados que requer um tratamento especial é o *type code*: um valor especial que indica algo particular a respeito de um tipo de instância. Isto geralmente aparece como enumerações, implementado como *static final integers*. Se os códigos são usados para informação e não alteram o comportamento da classe, pode-se usar *Substituição de Tipo de Código por Classe* que proporciona uma melhor verificação do tipo e uma plataforma para mover o comportamento posteriormente. Se o comportamento de uma classe for afetado pelo *type code*, deve-se usar *Substituição de Tipo de Código por Subclasses* caso isto seja possível. Caso contrário, pode-se usar *Substituição de Tipo de Código por Objeto do Tipo Estado/Estratégia* (Fowler, 2000a).

d) Simplificando Expressões Condicionais: O uso indevido de lógica condicional pode vir a tornar a compreensão do código um processo não trivial, por isso existe um conjunto de *refactorings* que podem ser aplicados com o objetivo de simplificá-los. O principal *refactoring* deste grupo é *Decomposição de Condicional* que partilha uma condicional em várias partes, sendo importante pelo fato de separar a lógica do chaveamento dos detalhes do que ocorrerá. Aplica-se *Consolidação de Expressão Condicional* quando tem-se vários testes e todos têm o mesmo resultado. Aplica-se *Consolidação de Trechos Duplicados em Expressão Condicional* para se remover qualquer duplicação pertencente ao trecho condicional do código. Usa-se *Substituição de Condicionais Encadeadas por Cláusulas* para tornar mais claros casos especiais de condicionais e *Remoção de Variável de Controle* para se eliminar *flags* de

controle. Programas orientados a objeto geralmente têm menor incidência de código condicional que programas procedurais, pois muito do comportamento condicional pode ser tratado através de polimorfismo. Polimorfismo torna-se mais adequado pelo fato de quem chama não precisar ter conhecimento do comportamento condicional, sendo também mais fácil a extensão das condições. Como conseqüência tem-se que programas orientados a objeto raramente necessitam efetivamente de sentenças condicionais com *switch*. Assim, caso seja detectado, haverá grandes possibilidades da aplicação de *Substituição de Condicional por Polimorfismo*. Polimorfismo também pode ser usado aplicando-se *Uso de Objeto Nulo* para remover verificações a respeito de valores nulos (Fowler, 2000a).

e) Tornando a chamada de métodos mais simples: Objetos estão diretamente relacionados a interfaces. Através de interfaces fáceis de compreender e usar pode-se atingir bons resultados em programação orientada a objetos. Na maioria das vezes, a forma mais simples e importante na atividade de avaliação dos métodos é a modificação do seu nome. Tendo-se o conhecimento do que um determinado programa está fazendo, o uso de *Renomeando Métodos* é indicado para que o próprio nome do método sirva como mecanismo para transmitir seus objetivos. Em relação a modificação dos nomes, o mesmo pode ser considerado para variáveis e classes, sendo neste caso mecanismos simples, não havendo portanto *refactorings* associados a tais atividades. Parâmetros têm grande importância no uso de interfaces, sendo o uso de *Adição de Parâmetros* e *Remoção de Parâmetros* bastante comum. Aqueles que são recentes em programação orientada a objetos geralmente usam lista de parâmetros longa, o que é típico de outros ambientes de programação. O uso de objetos permite que sejam usadas listas de parâmetros menores, existindo uma relação de *refactorings* cujo objetivo é a redução de tais listas. Se diversos valores de um objeto estiverem sendo usados, então pode ser aplicado *Uso de Objeto para Obtenção de Informações* para se reduzir tais valores a um único objeto. Caso este objeto não exista, pode-se criá-lo através de *Substituição de Parâmetro por Objeto*. Se for possível obter as informações de um objeto ao qual já se tenha acesso, podem ser eliminados parâmetros através de *Substituição de Parâmetro por Método*. No caso dos parâmetros serem usados para determinar comportamento condicional, pode-se usar *Substituição de Parâmetros por Métodos*. Vários métodos similares podem ser combinados, adicionando-se um parâmetro com *Parametrização de Método* . Deve haver uma separação clara entre métodos que podem modifcar o estado

de objetos daqueles que vão somente realizar consultas sobre o estado dos objetos. Caso esta separação não esteja sendo atendida, deve-se aplicar *Separação de Consulta do Modificador*. Boas interfaces devem deixar visível e disponível somente o necessário. Tanto as informações expressas através dos atributos como os métodos devem obedecer esta orientação anterior. Neste caso, deve-se aplicar *Ocultação de Método em Relação a Outras Classes* e *Remoção de Método de Atribuição de Valores*. Construtores pressupõem o conhecimento da classe de um objeto que será instanciado. Pode-se eliminar a necessidade deste conhecimento através de *Substituição de Construtor por Subclasses*. *Casting* é outro recurso que deve ser evitado através de *Encapsulando Projeção de Tipo*. Linguagens como Java possuem mecanismos de tratamento de exceção. Entretanto, aqueles que não têm familiriadade com tal recurso gerlamente usam códigos de erro para sinalizar a existência de problemas. Neste caso, deve-se usar *Substituição de Código de Erro por Exceção* para o uso dos recursos de exceção disponibilizados pela linguagem (Fowler, 2000a).

f) Trabalhando com Generalizações: associado a generalizações tem-se um conjunto de *refactorings*, sendo que a maioria deles trata da realocação de métodos ao longo da hierarquia. *Migração de Variável para Superclasse* e *Migração de Método para Superclasse* proporcionam a ascenção de membros das classes na hierarquia, enquanto que *Migração do Método para Subclasse* e *Migração de Variável para Subclasse* proporcionam o oposto. No caso de construtores a situação não é trivial, sendo necessária a aplicação de *Migração de Construtor para Superclasse* para lidar com este caso. No lugar de descer o construtor na hierarquia, deve-se usar *Substituição de Construtor por Subclasses*. Caso sejam detectados métodos que tenham estrutura parecidas, mas alguns detalhes diferentes, deve-se usar *Migração de Trecho de Código para um Novo Método Modelo* para que sejam separadas as diferenças. Além de promover a mudança de membros ao longo da hierarquia, a própria hierarquia pode ser modificada através da criação de novas classes. Migração de Trecho de Código para uma Nova Subclasse, Migração de Trecho de Código para uma Nova Superclasse e Migração de Trecho de Código para uma Nova Interface realizam esta tarefa. Caso seja verificado que existem classes desnecessárias na hierarquia, pode-se usar *Junção de Superclasse e Subclasse* para removê-los. Algumas vezes, percebe-se que herança não é a melhor forma de tratar uma determinada situação, sendo necessário o uso de delegação. Para este caso pode-se usar *Substituição de Herança por Delegação*. Para se

obter o resultado oposto pode-se usar Substituição de Herança por Delegação (Fowler, 2000a).

2.2.2. Classificando *refactorings* por nível de abstração

Em relação à classificação de *refactorings* por nível de abstração, têm-se os níveis procedural, orientado a objeto e conceitual conforme mostrado na Figura 2.3 (Norda, 2001).

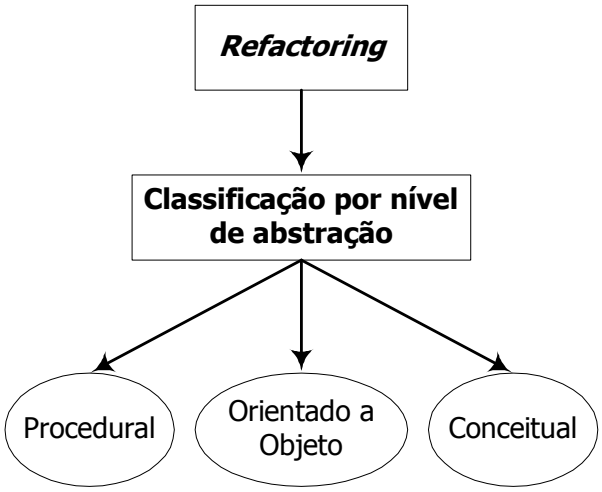


Figura 2.1: Classificação de Refactoring

Conforme apresentado na Figura 2.1, *refactorings* procedurais são de baixo nível e ocorrem em nível de código. Operações típicas são extração de método e controle da lógica para torná-la mais clara e objetiva. *Refactorings* orientados a objeto manipulam a estrutura do programa orientado a objeto, podendo ser exemplificado através da introdução de uma classe base abstrata ou movendo métodos ao longo da estrutura de hierarquia de classes. *Refactorings* conceituais focam na modificação ou no esclarecimento dos conceitos em um programa. Uma operação típica é a introdução de um padrão (Norda, 2001).

2.3. USANDO *REFACTORINGS*

O uso de *refactoring* pode ser representado nas seguintes etapas:

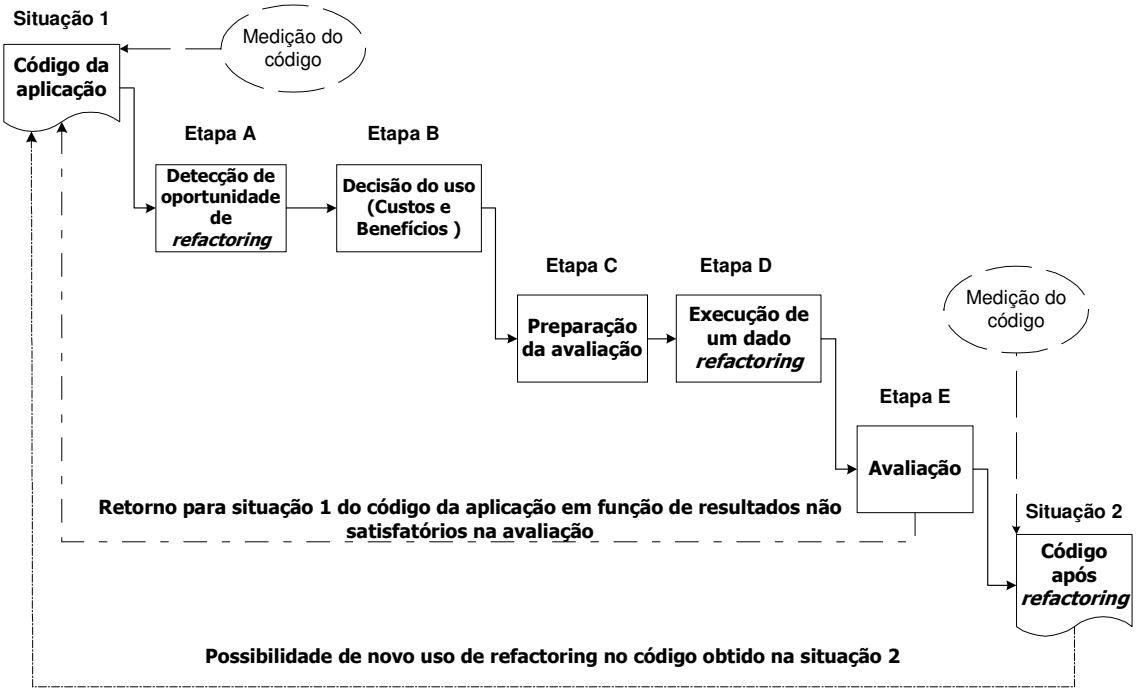


Figura 2.2: Etapas no uso de *refactoring*

a) **Etapa A - Detecção**: identificar trechos de código com oportunidades de *refactoring*.

b) **Etapa B - Análise da relação entre custos e benefícios**: qual a motivação e os benefícios decorrentes da aplicação do *refactoring*.

c) **Etapa C -Preparação da avaliação**: deve-se preparar o código para realização da avaliação antes da aplicação de *refactoring* (Fowler, 2000a). Nesta etapa e na de avaliação pode-se usar um *framework* de avaliação tal como o *Junit* (Beck e Gamma, 2000).

d) **Etapa D -Execução**: procedimentos usados para aplicar o *refactoring*.

e) **Etapa E -Avaliação**: uso de mecanismos para verificar a preservação de comportamento do programa.

Caso seja necessário desistir do processo em qualquer uma das etapas apresentadas na Figura 2.2, deve-se retornar para a situação 1. Uma vez na situação 2, pode-se prosseguir com a aplicação de outros *refactorings*. Entretanto, como condição para a aplicação de forma sucessiva por duas ou mais vezes, tem-se a preservação do comportamento da aplicação. O resultado de uma determinada seqüência de

refactorings somente será considerada como *refactoring* se for comprovada a preservação de comportamento da seqüência em questão (Cinnéide e Nixon, 2000). Opdyke (1992) estabeleceu que um programa deve possuir o mesmo comportamento observável externamente através de suas saídas para as entradas válidas antes e depois do *refactoring*, daí o mesmo autor usar o termo "preservação de comportamento observável". Esta dissertação não tem como objetivo verificar a preservação do comportamento observável através da aplicação de um ou mais *refactorings*. Assim, na realização dos estudos de casos apresentados no Capítulo 4, partiu-se do princípio que todos os *refactorings*, aplicados de forma isolada ou seqüencial, obedecem às caraterísticas de manutenção do comportamento observável.

Opdyke (1992) relata ainda que uma ferramenta destinada ao uso de *refactoring* pode assegurar que cada *refactoring* seja aplicado corretamente, não podendo, entretanto, decidir qual aplicar. Isto dificulta o uso de *refactorings* completamente automatizados, desde a detecção (Etapa A) até a aplicação (Etapa D) (Roberts, 1999). Assim, podem ser identificadas na literatura duas grandes limitações no uso de *refactoring*: detecção de oportunidades (Etapa A) e decisão de qual *refactoring* aplicar (Etapa B).

Neste contexto, esta dissertação propõe uma metodologia (Capítulo 3) e realiza dois estudos de caso (Capítulo 4) para relacionar *bad smells*, *refactorings* e métricas visando criar um conjunto de conhecimentos que auxilie na criação de mecanismos para superar tais limitações. Para tais fins, as etapas A e D foram consideradas preponderantes para o desenvolvimento desta dissertação.

2.3.1. Quando usar *refactoring*

Refactorings devem ser usados nas seguintes situações:

a) Antes de uma modificação ou adição de novas funcionalidades ao *software.* (Thompson e Reinke, 2001).

b) Após a obtenção de uma versão do *software* com o objetivo de torná-lo mais compreensível. (Thompson e Reinke, 2001).

2.3.2. Ferramentas disponíveis para o uso de *refactoring*

A Tabela 2.2 apresenta uma relação das principais ferramentas disponíveis no mercado para automatizar a aplicação de *refactoring*. Todas têm como pré-requisito a escolha do *refactoring* pelo usuário da ferramenta, ou seja, todas são limitadas quanto ao suporte à etapa A mencionada na Figura 2.2.

Tabela 2.2: Ferramentas para aplicação de *Refactoring*

Nome	Versão	Tipo
Chive Retool (Griffiths, 2001)	1.1.1	*Plugin* para Ambiente de Desenvolvimento Integrado -ADI (JBuilder)
CodeMorpher (CodeMorpher, 2002)	1.1	*Standalone* (*plugin* para ADI disponível somente em versão comercial)
Eclipse (Eclipse, 2002)	2.0.1	ADI
Intellij IDEA (Intellij, 2002)	2.5.1	ADI
JavaRefactor (JavaRefactor, 2002)	0.0.3	*Plugin* para editor (JEdit)
Jbuilder (JBuilder, 2003)	8	ADI
JFactor (JFactor, 2002)	1.1.4	*Plugin* para ADI (VisualAge)
Jrefactory (Jrefactory, 2000)	2.6.32	*Standalone* e *plugin* para ADI (Elixir, JBuilder)
RefactorIt (RefactorIt, 2002)	1.2.2	*Standalone* e *plugin* para ADI (Forte, JBuilder, Jdeveloper)
Together Control Center (TogetherControlCenter, 2002)	6.0	ADI
Transmogrif (Transmogrify, 2001)	m2	*Plugin* para ADI (JBuilder)
Xrefactory (Xrefactory, 2002)	1.5.10	*Plugin* para editor (Emacs, XEmacs)

2.4. DETECTANDO OPORTUNIDADES DE *REFACTORING*

A detecção de oportunidades de *refactoring* está associada com a Etapa A da Figura 2.2, podendo, em teoria, ser manual, semi-automática ou automática (Figura 2.3).

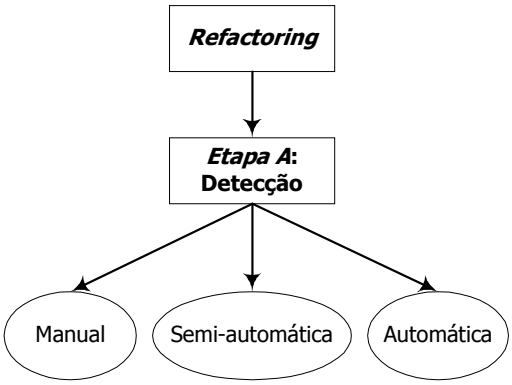


Figura 2.3: Detecção de oportunidades de *refactoring*

A forma manual depende exclusivamente da análise cognitiva para a seleção dos trechos com possibilidade de uso de *refactoring*. A forma semi-automática é caracterizada por possuir recursos adicionais que auxiliem a análise cognitiva. A localização automática de oportunidades de *refactoring* não é um processo trivial, pois tem como parâmetros percepções subjetivas do tipo do *bad smell* a serem eliminados (Emden e Moonen, 2002).

Nos itens a seguir serão apresentadas:

a) As formas de análise de programa mais usadas para detecção de oportunidade de *refactoring*.
b) Um levantamento realizado para esta dissertação com as principais alternativas para a detecção publicadas por grupos de pesquisa.

2.4.1. Análise Estática de programas

A análise das aplicações é importante para a obtenção das informações semânticas necessárias para a detecção de oportunidades de *refactoring* e para a decisão de qual *refactoring* aplicar (Norda, 2001). Na metodologia proposta nesta dissertação (Capítulo 3) foi considerada a análise estática no estudo de caso segundo a abordagem *bottom up*.

A análise estática examina o código fonte ou a imagem do executável da aplicação. Durante a análise estática o sistema não é executado. A análise estática é classificada de acordo com a técnica aplicada em análise estática quantitativa e análise estática qualitativa (Dudziak e Wloka, 2002).

A análise estática quantitativa determina valores numéricos através de métricas com base no código fonte ou imagem do executável da aplicação (Dudziak e Wloka, 2002).

A análise estática qualitativa pesquisa por características estruturais específicas baseada em regras pré-determinadas. Pode ser usada, por exemplo, para reduzir a quantidade de erros de codificação comuns. Uma ferramenta pode oferecer a funcionalidade para pesquisar um programa escrito com a linguagem em Java onde "==" está sendo usado para comparar objetos. Na realidade, o operador "==" é usado para avaliar a identidade de objetos, verificando se são os mesmos. Por outro lado, "*equals*" na linguagem Java possibilita determinar se os valores associados a objetos são os mesmos. Um outro exemplo de aplicação poderia ser a aplicação da regra: nunca dar acesso público a variáveis de instância ou de classe exceto quando a classe for uma estrutura de dados sem comportamento. Neste caso, podem ser verificadas e reportadas todas as ocorrências deste tipo de construção (Dudziak e Wloka, 2002).

2.4.2. Alternativas para a detecção de *refactoring*

Nossa pesquisa bibliográfica identificou as seguintes abordagens de detecção de oportunidades de *refactoring* propostas na literatura: detecção baseada em análise cognitiva (Fowler, 2000a); detecção baseada em Unified Modeling Language – UML (Astels, 2002); detecção baseada em meta-programação declarativa (Tourwé e outros, 2002), detecção baseada em invariantes (Ernst, 2000) (Kataoka e outros, 2001) e detecção baseada em métricas (Carneiro e Mendonça, 2002).

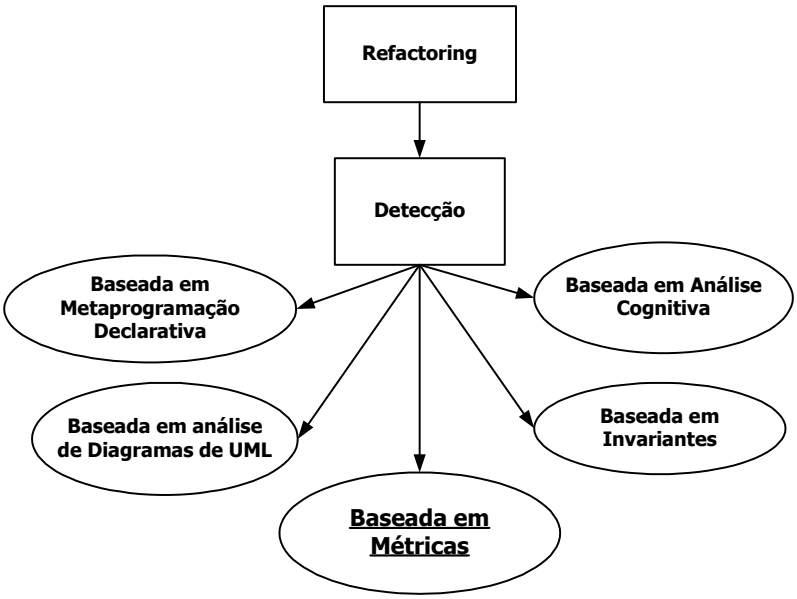


Figura 2.4: Tipos de detecção de refactoring

2.4.2.1. Detecção baseada em análise cognitiva

É baseada exclusivamente na experiência e conhecimento de quem efetua a análise. Folwer (2000a) exemplifica a análise da seguinte forma: "você deverá desenvolver seu próprio senso sobre o valor aceitável de variáveis de instância e o número máximo de linhas de código em um método". Esta alternativa de detecção pode vir a ser comprometida para situações de grande volume de código, além de não ser trivial o estabelecimento de parâmetros para indicar a eficiência e o tempo de realização da etapa de detecção no uso de refactoring (figura 2.2). Esta etapa é bastante dependente de aspectos subjetivos baseados no ser humano e de difícil automatização.

2.4.2.2. Detecção baseada em UML

A análise baseada em UML é um subconjunto da análise cognitiva, onde os artefatos de análise são baseados nos diagramas UML. A detecção de *refactoring* através de UML baseia-se em três princípios (Astels, 2002): por se tratar de uma representação gráfica, proporciona maior facilidade na visualização das classes e no seu relacionamento; torna possível a manipulação do código em um nível de maior granularidade (métodos, variáveis e classes no lugar de caracteres); a possibilidade de visualizar o código, especificamente o conteúdo das classes e o relacionamento entre as mesmas auxilia da detecção de *bad smells*.

2.4.2.3. Detecção baseada em meta-programação declarativa

Na detecção de *refactoring* através de meta-programação declarativa há a necessidade de declaração das orientações de projeto, possibilitando a pesquisa no código fonte do *software* para que seja verificado se o mesmo está seguindo tais orientações. Isto é possível através de um ambiente de metaprogramação declarativa tal como o SOUL (*Smalltalk Open Unifaction Language*) (Wuyts, 2001). Este tipo de ambiente é usado para descrever as regras do projeto como meio para detectar violações das mesmas. As regras de projeto devem ser seguidas durante todas as fases de ciclo do *software*. No caso do SOUL, usa-se uma linguagem de programação lógica com forte integração com o ambiente de desenvolvimento (Wuyts, 2001). A linguagem lógica é usada para criar uma metalinguagem de avaliação da implementação. Desta forma, é possível a escrita de regras lógicas para avaliar aplicações orientadas a objeto, independentemente da linguagem utilizada no nível base. Assim, as mesmas regras lógicas para a extração de informações a respeito do código fonte do *software* podem ser aplicadas para diferentes linguagens orientadas a objeto (Fabry e Mens, 2003).

2.4.2.4. Detecção baseada em invariantes

Trabalhos têm demonstrado a viabilidade de detecção automática de trechos de código de um programa candidatos a determinados tipos de *refactorings* através de invariantes de programas: quando um padrão particular de relacionamento de invariante aparece em determinada parte do código, um *refactoring* específico é aplicável. Sendo muitos programas carentes de invariantes explícitas, uma ferramenta de detecção de invariante chamada Daikon (Ernst, 2000) foi proposta para deduzir invariantes. Os criadores da ferramenta desenvolveram um padrão de detecção de invariante para vários tipos comuns de *refactoring*, aplicando-se os mesmos em um código base Java (Ernst e outros, 2001).

Kataoka e sua equipe (2001) propuseram uma ferramenta para a identificação de candidatos a *refactoring*. Com o objetivo de enfatizar a vantagem da aplicação de *refactoring* proporcionado pela redução do custo de detecção de candidatos e escolha daquele que deve seja mais adequado, a abordagem proposta utiliza invariantes detectadas pela ferramenta *Daikon* (Ernst e outros, 2001) para identificar condições favoráveis para a aplicação de *refactoring* em um processo de duas etapas. A primeira consiste no uso da ferramenta *Daikon* (Ernst, 2000) para realizar a detecção dinâmica de

invariantes no programa em questão através do rastreamento das variáveis de interesse, aplicando um conjunto de testes ao programa e obtendo as variantes através dos valores obtidos dos testes. Na segunda etapa, o programa e as invariantes localizadas anteriormente são consideradas como referência para a identificação de candidatos a *refactoring*. Baseando-se no catálogo de *refactorings* proposto por Fowler (2000a), foi desenvolvido um detector de padrões para os *refactorings* mais comuns. A ferramenta *Daikon*, usada para detectar as invariantes, em conjunto com o detector de padrões compõem o conjunto de ferramentas para identificar candidatos a *refactoring* (Carlos, 2002).

2.4.2.5. Detecção baseada em métricas

A metodologia proposta por Carneiro e Mendonça (2002) introduz métricas no uso de *refactoring*. Conforme será abordado de forma mais detalhada no Capítulo 3 com a apresentação da metodologia e no Capítulo 4 com a exposição dos estudos de caso, são usadas duas abordagens (*Top Down*, baseada no paradigma Meta Pergunta Métrica e *Bottom Up*, baseada na medição do código ao longo da seqüência de *refactorings*) com o propósito de estabelecer relacionamentos entre *bad smells*, *refactorings* e métricas. Os autores argumentam que métricas podem ser utilizadas para facilitar a etapa de detecção prevista no uso de *refactoring* (conforme apresentado na Figura 2.2), mesmo que na última etapa seja aplicada análise cognitiva para dirimir situações previstas pelas falsas positivas (falsas suspeitas de *bad smells* proporcionadas pelas métricas, mas que podem ser descartadas através de análise cognitiva) e falsas negativas (*bad smells* não detectados pelas métricas, mas que de fato existem e que podem ser detectados através de análise cognitiva).

Parte-se da premissa que as vantagens do uso de métricas superam os custos associados às falsas positivas e falsas negativas (Carneiro e Mendonça, 2002), sendo a grande vantagem da metodologia a redução do volume de código fonte a ser analisado.

2.5. CONCEITOS BÁSICOS DE MEDIÇÃO

A melhoria do processo e dos produtos de *software* só ocorrerá quando no projeto forem definidos claramente os objetivos dos mesmos. A melhor forma de se definir claramente estes objetivos é a forma quantitativa (com números ao invés de palavras), daí a necessidade do uso de medição. Uma medida será útil se colaborar para

a compreensão de um processo em estudo ou um dos seus produtos resultantes. Não se deve medir somente aquilo que é conveniente ou fácil de medir, mas aquilo que será necessário para que sejam atingidos os objetivos previamente estabelecidos (Briand, 1999). Com base nesta premissa, esta dissertação estuda o relacionamento entre métricas e *refactorings*. Esta seção discute o conceito de medição e lista as principais métricas difundidas na engenharia de *software*.

2.5.1. Mensuração e métricas em engenharia de software

Métricas têm sido usadas tradicionalmente durante o processo de desenvolvimento de *software* com o objetivo de melhorar a qualidade tanto do processo de desenvolvimento como do produto. Para exemplificar, métricas podem ser usadas para mensurar a complexidade do projeto do software, fornecer previsões a respeito das propriedades do produto final ou do custo total de desenvolvimento. Métricas também desempenham importante papel no processo de engenharia reversa, podendo ser aplicadas, por exemplo, para avaliar a complexidade do código. Problemas de projeto também podem ser identificados através de métricas apropriadas. Métricas especializadas na interação entre objetos podem revelar alto acoplamento e baixa coesão em determinadas partes do software. O acoplamento dificulta a modificação e o reuso do código. Indícios de baixa coesão também precisam ser analisados, uma vez que tal característica em uma classe sugere que as variáveis e os métodos da mesma não estão sendo usados de forma apropriada que justifique a sua existência. Métricas associadas à hierarquia de herança de sistemas orientados a objeto podem ser usadas para avaliação do nível de reuso do software. Algumas métricas podem ser aplicadas em sistemas escritos em qualquer linguagem, enquanto outras são dependentes do paradigma de programação ou da linguagem em questão. Por exemplo, métricas orientadas a objeto são usadas para avaliar sistemas orientados a objeto (Systä, 2000).

Neste contexto, as métricas de software definem atributos de qualidade do software que podem ser usados para auxiliar nas diversas etapas do uso de *refactoring*, conforme descrito na Seção 2.3. A mensuração de código auxilia no julgamento a respeito das características de uma dada aplicação, sendo que o seu resultado, ainda que incompleto, proporciona informações de grande potencial para a avaliação das características do *software*.

2.5.2. O Paradigma Meta Pergunta Métrica

O paradigma Meta Pergunta Métrica - MPM (*Goal Question Metric* – GQM) foi proposto por Basili (Basili e Weiss, 1984) como uma abordagem para medição de *software*.

Segundo este paradigma, devem ser seguidas três etapas para se identificar o que será efetivamente medido (Mendonça, 1997) (Sölingen e Berghout, 1999):

1) Enumerar os principais objetivos do desenvolvimento ou manutenção do projeto: é a definição das metas de medição direcionadas para determinadas necessidades específicas. No estudo de caso usando a abordagem *Top Down* apresentado no Capítulo 5 desta dissertação foi adotado o mesmo ponto de vista do Fowler para a definição das metas. Vale ressaltar, que outras metas podem ser definidas a partir de um mesmo objeto de estudo, mas com outros pontos de vista. O modelo a seguir é usado para definir as metas da medição (Basili e Rombach, 1988).

Analisar 'objeto de estudo' com o objetivo de 'finalidade' com respeito ao 'foco' do ponto de vista do 'ponto de vista', onde:

- Objeto de estudo: uma entidade da organização para o usuário do modelo.
- Finalidade: representa o porquê da medição com palavras como caracterizar, avaliar, controlar, melhorar, prever, entre outras.
- Foco: atributo principal que se deseja medir.
- Ponto de vista: qualquer dado do grupo de usuários.

2) A partir de cada meta, elaborar perguntas para que seja verificado se as mesmas estão sendo alcançadas: as metas definidas são mapeadas para perguntas que devem ser respondidas.

3) Decidir o que deve ser medido para que seja possível responder às questões adequadamente: as métricas são usadas para quantificar as respostas.

Quando do mapeamento das perguntas para as métricas já existentes, podem ocorrer as seguintes situações (Mendonça e Basili, 2000):

- Nenhuma das métricas disponíveis atender às necessidades de um determinado foco (um dos *bad smells*, conforme será apresentado na Seção 3.1), pois as mesmas ainda não estão difundidas e validadas.
- Uma métrica atendendo às necessidades.
- Duas ou mais métricas atendendo às necessidades.

A etapa de mapeamento das perguntas para as métricas é de essencial importância para o sucesso da aplicação do paradigma MPM. Por este motivo, deve-se assegurar que as medidas obtidas realmente capturem as informações necessárias e desejáveis de um conjunto de atributos que se deseja medir.

Este paradigma tem sido usado de forma bem sucedida em diversas organizações: NASA, Motorola, HP e IBM (Mendonça, 1997).

A Figura 2.5 mostra um exemplo abstrato da estrutura do paradigma Meta Pergunta Métrica (Mendonça e outros, 1998).

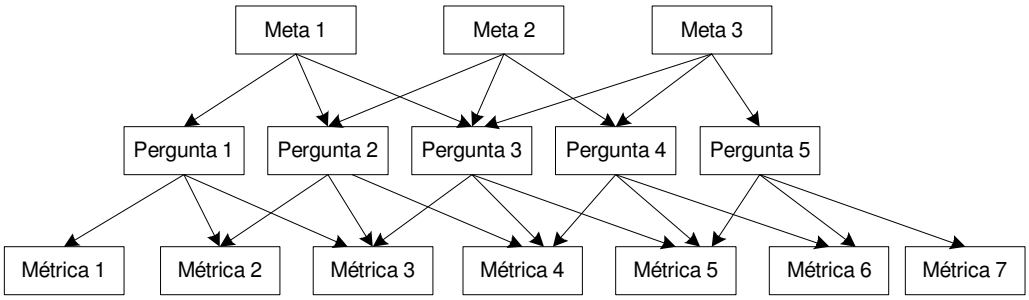


Figura 2.5: Estrutura abstrata do paradigma Meta Pergunta Métrica

2.5.3. Algumas das principais métricas difundidas na literatura

Esta seção apresentará algumas das principais métricas definidas na literatura que podem ser aplicadas ao código fonte de software orientado a objeto.

Existem muitas métricas aplicadas para a medição de aplicações tradicionais não orientadas a objetos. Duas destas métricas são também muito utilizadas em aplicações orientadas a objetos, estando associadas a tamanho (Fenton e outros, 1997) e complexidade (McCabe, 1976).

Como exemplos de métricas orientadas a objetos têm-se as métricas propostas por Chidamber e Kemerer (Chidamber e outros, 1994) e Lorenz (1994). As mesmas podem ser agrupadas em três categorias: métricas de herança, métricas de complexidade e métricas de comunicação. O primeiro grupo mede o uso de herança e a implementação de hierarquia no software, o segundo mede a complexidade, enquanto que o terceiro mede o acoplamento e a coesão entre as classes.

Será apresentada na Tabela 2.3 um resumo com as principais métricas utilizadas e o que as mesmas buscam medir.

Tabela 2.3. Métricas difundidas na literatura

Métrica	Aplicada em	Atributo	O que medem
Cyclomatic complexity - CC (McCabe, 1976)	Classes e métodos	Complexidade	Complexidade e alternativas possíveis no controle de fluxo
Lines of Code – LOC (Lorenz e outros, 1994)	Classes e Método	Tamanho	Obter o tamanho das classes e métodos.
Depth of Inheritance Tree - DIT (Chidamber e outros, 1994)	Classes e interfaces	Herança	Reuso, compreensão e teste
Number of Children -NOC (Chidamber e outros, 1994)	Classes e interfaces	Herança	Reuso
Response for Class -RFC (Chidamber e outros, 1994)	Classes	Comunicação	Acoplamento, complexidade e pré-requisitos para teste
Coupling between object classes - CBO (Chidamber e outros, 1994)	Classes	Comunicação	Coesão e reuso
Lack of Cohesion in Methods - LCOM (Chidamber e outros, 1994)	Classes	Comunicação	Coesão, complexidade, encapsulamento e uso de variáveis
Weighted Methods per Class - WMC (Chidamber e Kemerer, 1994)	Classes	Complexidade	Complexidade, tamanho, esforço para manutenção e reuso.
Number of Methods - NOM[1] (Lorenz e outros, 1994)	Métodos	Tamanho	Corresponde a WMC onde o peso de cada método é 1.
Number of Statements – NOS (Lorenz e outros, 1994)	Métodos	Tamanho	Obter o número de sentenças em um método.
Number of Instance Variables – NIV (Lorenz e outros, 1994)	Classes	Tamanho	Obter o número de variáveis de instância.
Number of Class Variables – NCV (Lorenz e outros, 1994)	Classes	Tamanho	Obter o número de variáveis de classe.
Number of Inherited Methods – NMI (Lorenz e outros, 1994)	Métodos	Herança	Obter o número de métodos herdados e definidos em uma superclasse.
Number of Overriden Methods – NMO (Lorenz e outros, 1994)	Métodos	Herança	Obter o número de métodos definidos em uma superclasse e redefinidos na subclasse.

Esta dissertação mostra que métricas podem auxiliar nas etapas do uso de *refactoring,* notadamente na etapa A da Figura 2.2*,* desde que complementadas com análise cognitiva. No Capítulo seguinte será apresentada uma metodologia baseada em análise da estrutura estática do código da aplicação para incluir métricas no uso de *refactoring*.

[1] Derivação da métrica *Weighted Methods per Class* (WMC) proposta em (Chidamber e Kemerer, 1994).

3. METODOLOGIA PROPOSTA

A abordagem cognitiva proposta por Fowler (2000a) é mostrada na Figura 3.1. Nesta dissertação, buscamos estabelecer relacionamentos entre métricas, *refactorings* e *bad smells* usando a abordagem de Fowler como arcabouço de partida e posterior uso das seguintes abordagens:

a) Uma abordagem *Top Down* usando o paradigma Meta Pergunta Métrica - MPM (Basili e Weiss, 1984).

b) Uma abordagem *Bottom Up* usando análise empírica. Esta segunda abordagem utiliza um grande conjunto de métricas e verifica o seu relacionamento com *refactorings* e *bad smells*. Este conjunto de métricas é referenciado nesta dissertação pela expressão "métricas disponíveis".

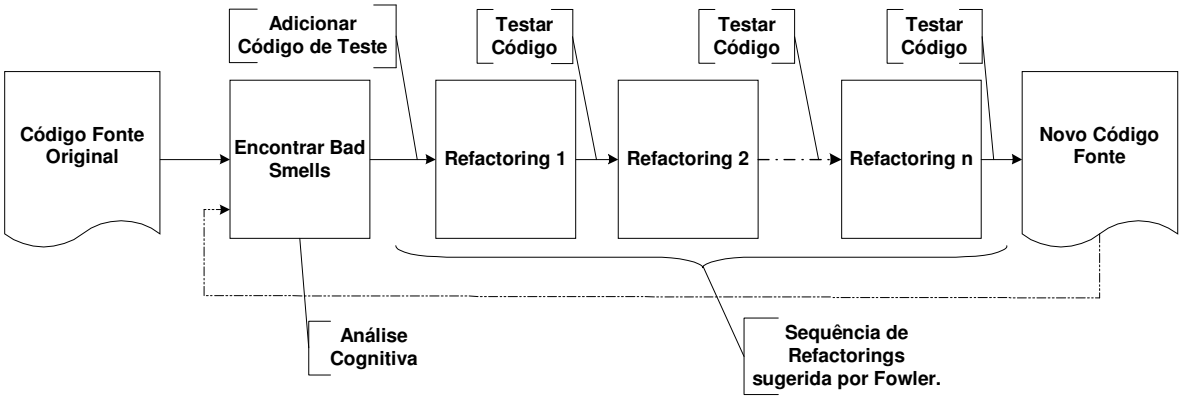


Figura 3.1: Aplicação tradicional de *refactoring* proposto por Fowler através de metodologia cognitiva

3.1. RELACIONANDO MÉTRICAS E *REFACTORING* NA ABORDAGEM *TOP DOWN*

Esta abordagem tem o objetivo de indicar métricas adequadas à detecção dos bad smells propostos por Fowler (2000a).

Conforme apresenta a Figura 3.2, nossa metodologia usa o catálogo de bad smells apresentado por Fowler (2000a) como ponto de partida para definição de metas e, em seguida, busca identificar as perguntas e as métricas mais adequadas a estas metas. O modelo aplicado, baseando-se no paradigma MPM, mapeia bad smells para métricas conhecidas ou métricas novas ainda não difundidas na literatura.

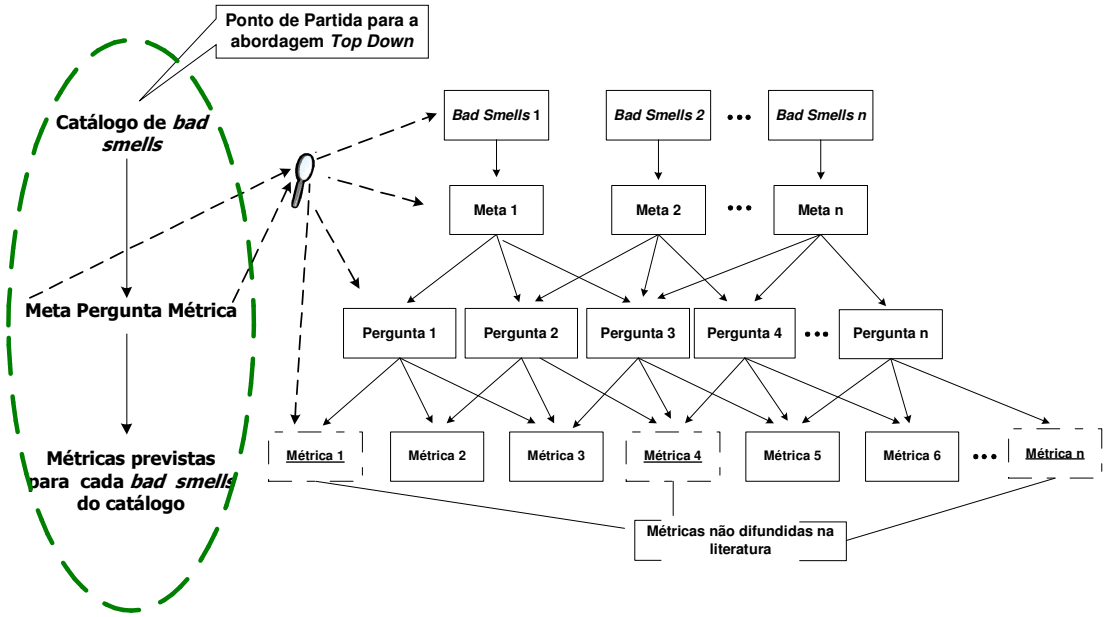


Figura 3.2: Introdução de métricas no uso de *refactoring* através de uma abordagem *Top Down* baseando-se no paradigma Meta Pergunta Métrica

Na Seção 4.1.2 do Capítulo 4 é apresentada a aplicação da abordagem *Top Down* conforme descrito na Figura 3.2 para cada um dos *bad smells* sugeridos por Fowler (2000a).

3.2. RELACIONANDO MÉTRICAS E *REFACTORING* NA ABORDAGEM *BOTTOM UP*

Para complementação da abordagem *Top Down* proposta, tem-se a abordagem *Bottom-Up* baseada em análise empírica. Esta abordagem tem os seguintes objetivos conforme apresentado na Figura 3.3:

a) Verificar se o que foi previsto na abordagem *Top-Down* ocorreu, isto é, se os valores das métricas derivadas realmente sofreram variações representativas.
b) Verificar ocorrências não previstas: métricas que variaram de forma inesperada.
c) Identificar quais as métricas foram previstas e não ocorreram.
d) Fornecer dados concretos para a expansão do número de heurísticas propostas por Demeyer em (Demeyer e outros, 2000). Vide Seção 3.4.1.

Na abordagem *Bottom Up* considera-se um grande universo de métricas (vide Apêndice 01), sendo verificadas quais dessas métricas foram alteradas em função dos *refactorings* aplicados, assim como os *bad smells* correspondentes.

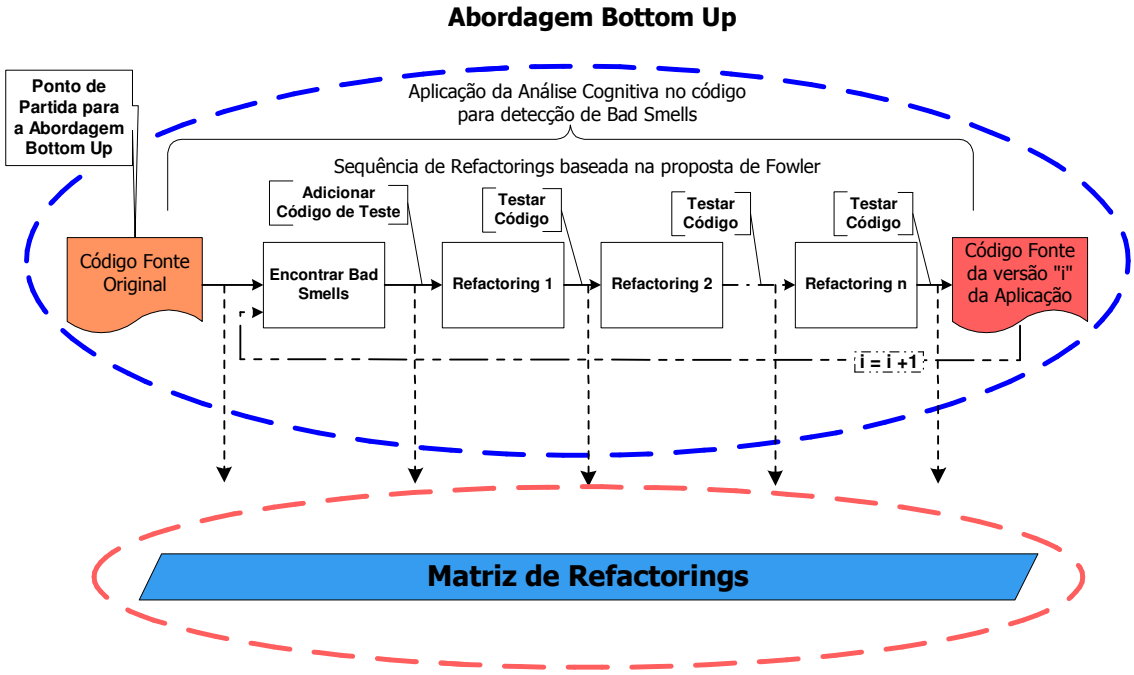


Figura 3.3: Abordagem *Bottom Up* e Matriz de *Refactoring*

3.2.1. Matriz de *Refactorings*

Para fazer a análise da variação das métricas, foi concebido o que chamamos de matriz de *refactorings* (Figura 3.4). A matriz de *refactorings* tem o objetivo de fornecer um painel informativo de medições ao longo da seqüência de *refactorings* e entre os *bad smells*. Esta matriz possibilita registrar o histórico dos *bad smells* detectados para um *software*, assim como os respectivos *refactorings* usados para eliminá-los. Na matriz, cada coluna é exclusiva para o armazenamento de informações relacionadas a um determinado *bad smells*. Após o uso de um dado *refactoring*, faz-se o preenchimento da célula da matriz com as seguintes informações: nome do *refactoring*, classificação por objetivo em que se enquadra e referência para o local onde devem ser encontrados os resultados das medições após o *refactoring*. A seguir são destacados os motivos da concepção e uso da matriz de *refactorings*:

a) Estabelecimento de parâmetros para avaliação do efeito de *refactorings* sobre a aplicação.

b) Estabelecimento de parâmetros para avaliação do efeito de um conjunto de *refactorings*, necessários para eliminar *bad smells*, sobre a aplicação.

c) Estabelecimento de parâmetros para comparar a eficiência das abordagens *top down* e *bottom up*.

d) Possibilidade de acompanhamento das medidas da aplicação durante a resolução de um *bad smells* e entre *bad smells* consecutivos ou não.

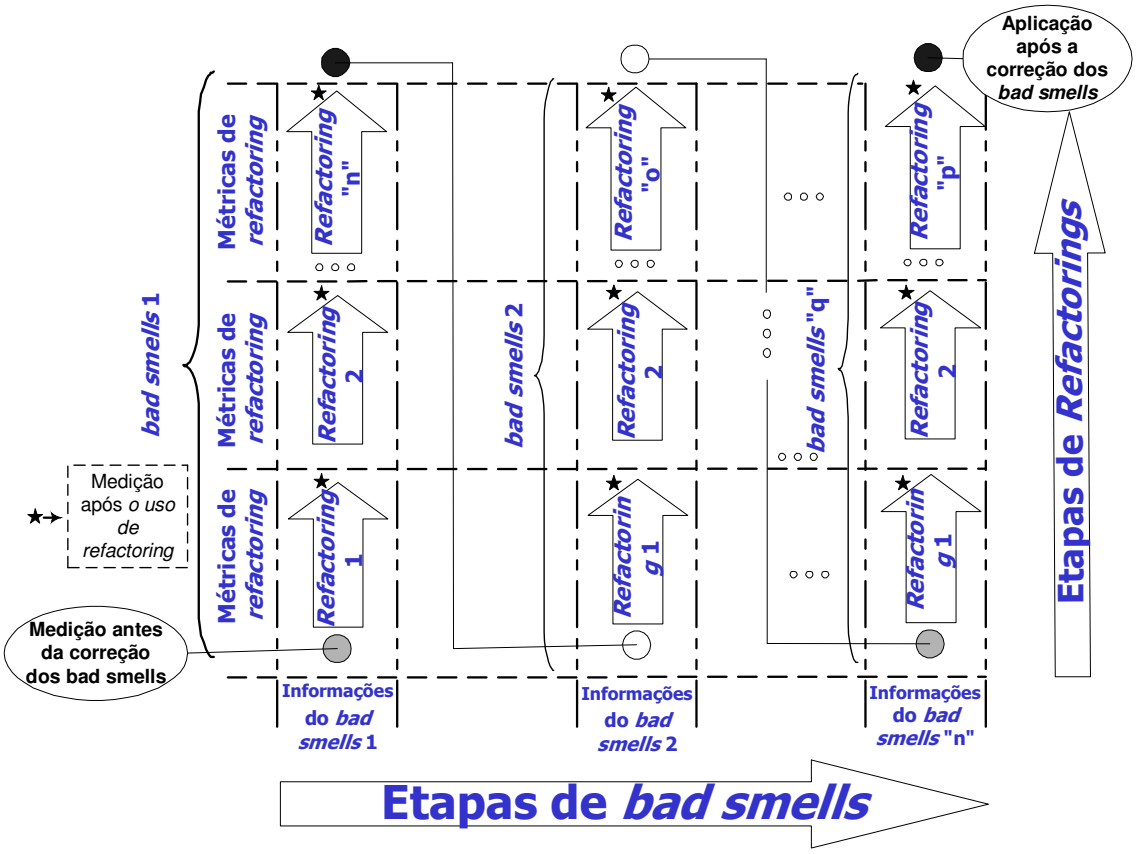


Figura 3.4: Matriz de r*efactorings*

3.3. COMPARANDO AS ABORDAGENS *TOP DOWN* E *BOTTOM UP*

A abordagem *Top Down* parte de cada *bad smells* separadamente para um conjunto de métricas obtidos através do paradigma Meta Pergunta Métrica. Este conjunto de métricas tem o objetivo de indicar a presença de um determinado *bad smells*. Não faz parte da proposta indicar qual a seqüência de *refactorings* a ser aplicada, nem qual o comportamento do *software* em termos de medidas ao longo do processo de execução da seqüência de *refactorings*.

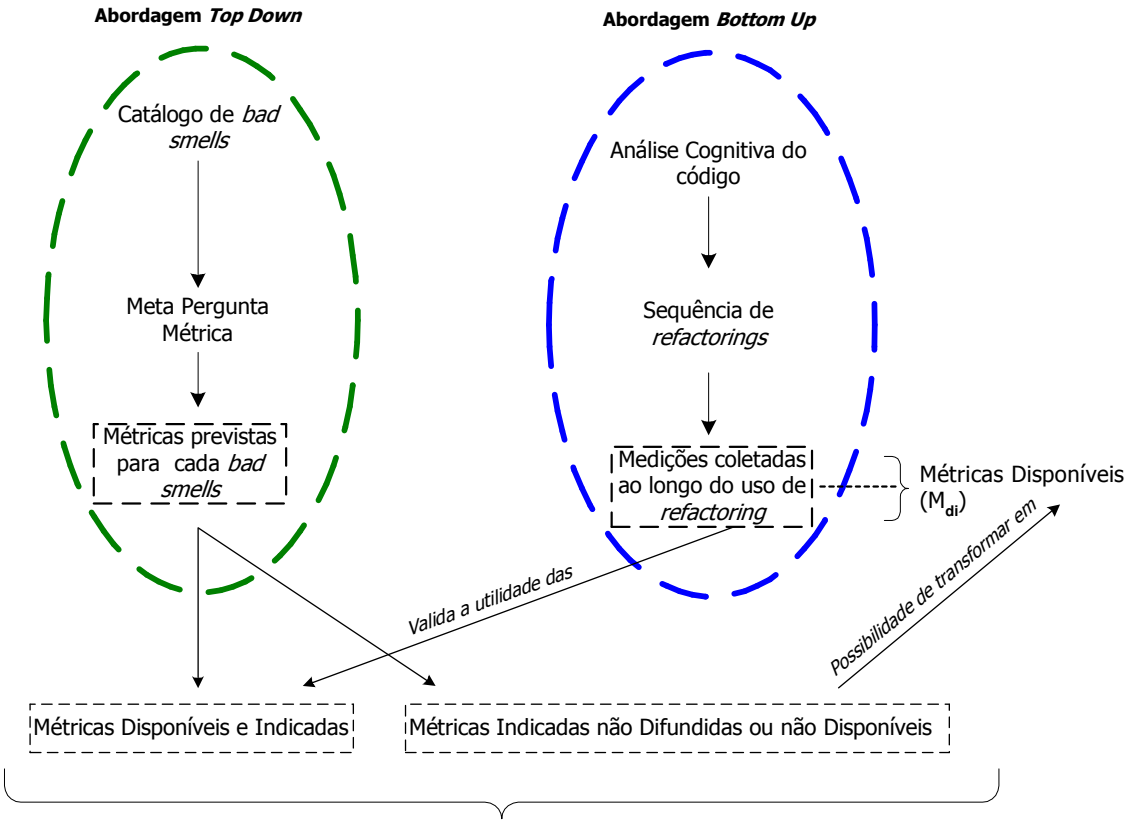


Figura 3.5: Relacionamento entre as abordagens *Top Down* e *Bottom Up*

Na abordagem *Bottom Up*, faz-se a verificação não somente a nível de *bad smells*, mas também a nível de *refactoring*. Para esta finalidade é usada a matriz de *refactorings*, onde são consideradas duas dimensões: uma associada ao *bad smells* e a outra associada aos *refactorings,* usando-se para análise um conjunto previamente conhecido de métricas. Desta forma, um conjunto de medidas, mais amplo, é associado aos *bad smells*, enquanto outro conjunto, mais reduzido, é associado aos *refactorings* específicos. A matriz de *refactorings* possibilita que se verifique se as métricas indicadas na abordagem *Top Down* estão coerentes com os *bad smells* e os *refactorings* apresentados por Fowler (2000a).

As métricas derivadas na abordagem *Top Down* e que ainda não estão difundidas ou disponíveis podem eventualmente vir a fazer parte do conjunto de métricas utilizado na abordagem *Bottom Up*, desde que devidamente definidas, implementadas e testadas.

3.3.1. Análise da eficiência das abordagens

A abordagem *Top Down* baseada no paradigma Meta Pergunta Métrica (Basili e Weiss, 1984) tem como resultado um conjunto de métricas cujo objetivo é responder as perguntas formuladas em função das metas estabelecidas. Este conjunto, conforme já mencionado na Seção 2.5.2, pode conduzir às seguintes situações:

- Nenhuma das métricas disponíveis atende às necessidades de um determinado foco (um dos *bad smells*, conforme foi apresentado na Seção 3.1), pois as mesmas ainda não estão difundidas e validadas.
- Uma métrica atende às necessidades.
- Duas ou mais métricas atendem às necessidades.

Assim, serão consideradas nesta dissertação as seguintes classificações para métricas:

a) Métricas Indicadas não Difundidas ou não Disponíveis: são indicadas através da abordagem Meta Pergunta Métrica, mas não fazem parte do conjunto de métricas disponíveis, seja porque não são difundidas na literatura, seja porque ainda não estão disponíveis para uso na ferramenta de medição adotada.

b) Métricas Disponíveis: métricas disponibilizadas pela ferramenta de medição adotada. Podem ser classificadas ainda como:

   b.1) Métricas Disponíveis e Indicadas: métricas disponibilizadas pela ferramenta de medição adotada e indicadas pela abordagem *Top Down*.

   b.2) Métricas Disponíveis e não Indicadas: métricas disponibilizadas pela ferramenta de medição adotada e não indicadas pela abordagem *Top Down*.

A principal vantagem da abordagem *Top Down* é a possibilidade de identificação das métricas indicadas não difundidas ou não disponíveis, fato que a torna independente do conjunto de métricas que será trabalhado na abordagem *Bottom Up*. Além disso, esta abordagem traz a possibilidade da derivação de métricas ainda não difundidas para auxílio no uso de *refactoring*. As métricas ainda não difundidas podem ser definidas, implementadas, validadas e usadas posteriormente na abordagem *Bottom Up* para a verificação da eficiência da primeira abordagem e melhora no uso de *refactoring*. Em relação ao conjunto de métricas disponíveis tem-se que

Mdi = Mdi(i) $\cup$ Mdi(ni), (3.1)

onde

Mdi : Métricas Disponíveis

Mdi(i) : Métricas Disponíveis e Indicadas

Mdi(ni) : Métricas Disponíveis e Não Indicadas

3.4. METODOLOGIA RESULTANTE DAS DUAS ABORDAGENS

O trabalho iniciado nesta dissertação tem o objetivo de estabelecer mecanismos de relacionamento entre métricas, *refactorings* e *bad smells* que sirvam de base para a validação da metodologia apresentada na Figura 3.6.

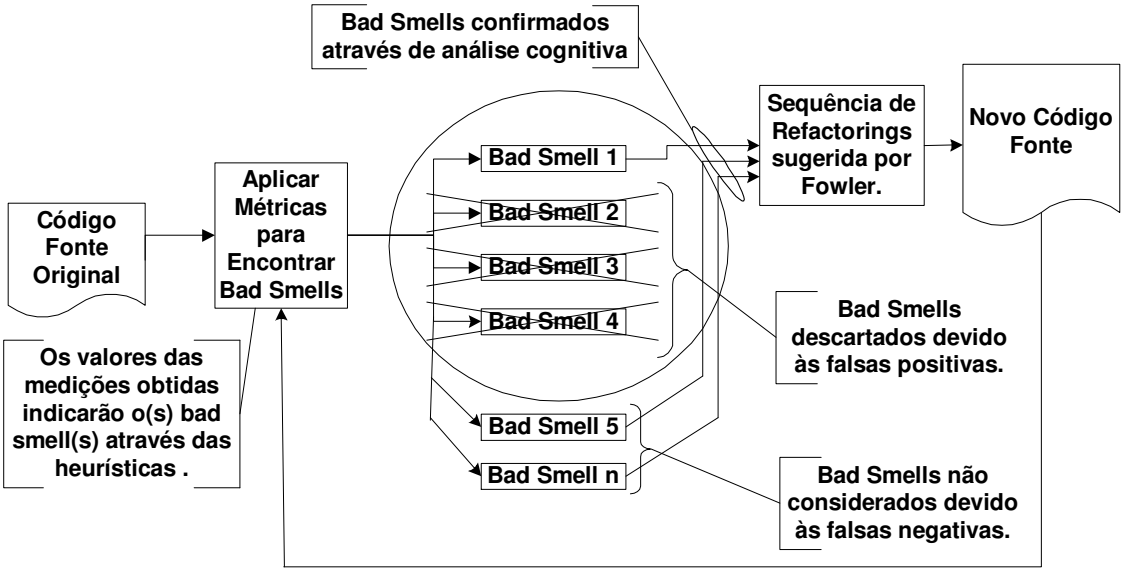


Figura 3.6: Aplicação de *refactoring* usando a metodologia proposta

Esta metodologia proposta consiste em:

a) Aplicar a medição de um conjunto de métricas pré-definido no código fonte da aplicação.
b) Associar os resultados obtidos com os *bad smells*.
c) Efetuar análise cognitiva para verificar a necessidade de aplicação de um ou mais *refactorings* ou a ocorrência de falsas positivas ou negativas.
d) Aplicar um ou mais *refactorings* associados aos *bad smells* indicados por Fowler (2000a).

A metodologia proposta indica, portanto, como associar as medições obtidas com os *bad smells*. A grande vantagem da metodologia apresentada é reduzir o volume de código fonte a ser analisado pela equipe de *refactoring*.

Um ponto importante da metodologia é o uso da abordagem MPM para indicar métricas novas para quantificar certos *bad smells*. Isto permite que pesquisadores da área de medição de software utilizem as heurísticas desenvolvidas por pesquisadores da área de *refactoring* na identificação de novas métricas para avaliação do código. Além disso, cria-se uma possibilidade de relacionamento entre a análise cognitiva tradicionalmente feita antes do *refactoring* e a avaliação empírica possibilitada pelas métricas utilizadas para quantificar os *bad smells*.

É importante ressaltar que o uso de métricas cria dois tipos de riscos: a indicação de falsas positivas e as falsas negativas como apresentadas na Figura 3.6. No caso das falsas positivas, as métricas conduzirão a falsas suspeitas de *bad smells* que deverão ser descartadas através de análise cognitiva. Já as falsas negativas são *bad smells* não detectados pelas métricas, mas que de fato existem e que poderiam ser detectados através de análise cognitiva. Nosso trabalho parte da premissa que as vantagens do uso de métricas superam os custos associados às falsas positivas e falsas negativas quando considedrada a possibilidade da redução do código fonte a ser analisado.

3.4.1. Abordagens Similares

Como resultado de levantamento bibliográfico realizado, verificou-se a existência de outra abordagem que relaciona métricas de código e *refactoring*. Essa abordagem propõe heurísticas para a identificação de *refactorings* já executados entre diferentes versões de uma dada aplicação (Demeyer e outros, 2000), limitando-se a somente cinco *refactorings*: Migração de Trecho de Código para uma Nova Superclasse, Dissolução de Superclasse (*Merge with Superclass*), Migração de Trecho de Código para uma Nova Subclasse, Dissolução de Subclasse (*Merge with Subclass*) e Migração de Trecho de Código para outra Classe (*Move to Other Class*). Sendo que destes cinco refactorings, três não constam no catálogo apresentado por Fowler em (2000a): Dissolução de Superclasse, Dissolução de Subclasse e Migração de Trecho de Código para outra Classe. A abordagem aqui apresentada fornece um conjunto de dados que possibilita ampliar o escopo da análise para outros *refactorings*, além da possibilidade de relacioná-los com *bad smells*.

Ao final deste capítulo é importante ressaltar que apesar da metodologia proposta na Figura 3.6 ser uma das principais contribuições deste trabalho, esta dissertação tem como foco um problema menor: o estudo do relacionamento entre métricas, *refactorings* e *bad smells* usando as abordagens descritas nas Seções 3.1 e 3.2. Esta dissertação é um primeiro passo no sentido de criar e validar a metodologia apresentada na Figura 3.6. Vários outros passos são necessários para o uso efetivo desta metodologia. Eles são descritos na conclusão deste trabalho, conforme apresentado no Capítulo 5.

4. ESTUDO DE CASO

Para avaliar as abordagens propostas nas Seções 3.1 e 3.2, foram feitos estudos de caso da sua aplicação. Os estudos de caso são descritos a seguir, sendo realizado um para cada abordagem. O primeiro estudo de caso baseia-se em um processo analítico que tem como ponto de partida o catálogo de *bad smells* apresentado em (Fowler, 2000a) e como resultado conjuntos de métricas derivadas através do paradigma Meta Pergunta Métrica. O segundo estudo de caso é notadamente empírico, pois usa um conjunto amplo de métricas para investigar o seu relacionamento com *bad smells* e respectivos *refactorings* utlizados para corrigí-los em uma aplicação sugerida por consulta via correio eletrônico ao próprio Fowler.

4.1. ESTUDO DE CASO USANDO A ABORDAGEM *TOP DOWN*

A abordagem *Top Down* foi utilizada neste estudo de caso para indicar/derivar métricas adequadas de 16 dos 22 *bad smells* propostos por Fowler (2000a).

4.1.1. Descrição do estudo

Conforme descrito na Seção 3.1, o catálogo de *bad smells* é o ponto de partida para a definição de metas. Foram utilizadas a descrição, as características e os *refactorings* associados aos *bad smells* apresentados por Fowler (2000a) para a elaboração dos seguintes itens que compõem o estudo de caso segundo a abordagem *Top Down*:

a) Nome do *bad smells.*
b) Descrição.
c) *Refactorings* associados ao *bad smells.*
d) Possibilidades mais prováveis de ocorrência dos *bad smells.*
e) Estabelecimento da meta segundo a abordagem Meta Pergunta Métrica (MPM).

   Uso da abordagem MPM no *bad smells* para a derivação do conjunto de métricas associado a cada pergunta formulada. Este conjunto pode ser composto por uma ou mais métricas disponíveis ou ainda por métricas não difundidas conforme descrito a seguir:

   - Tipo 1: Métricas disponíveis e indicadas para auxílio no processo de análise cognitiva para obtenção do resultado desejado.
   - Tipo 2: Métricas disponíveis e com possibilidade de derivação de novas métricas a partir do conjunto obtido.

- Tipo 3: Métricas não disponíveis ou não difundidas, mas que podem ser obtidas através de análise estática qualitativa.
- Tipo 4: Métricas não disponíveis ou não difundidas a serem derivadas a partir de uma ou mais métricas pertencentes ao conjunto de métricas obtido como resposta de outra pergunta.
- Tipo 5: Métricas não disponíveis ou não difundidas, cujo resultado depende fortemente de análise cognitiva, sendo, portanto, de difícil implementação.
- Tipo 6: Métricas não disponíveis ou não difundidas, mas que podem ser obtidas através de análise dinâmica.

4.1.2. Resultados

Será apresentado a seguir o conjunto de resultados obtidos para os 16 dos 22 *bad smells* propostos por Fowler (2000a) com o formato descrito na Seção 4.1.1.

4.1.2.1. Código Duplicado

Considerado o mais comum dos *bad smells*. Deve-se sempre encontrar uma forma de eliminar a repetição do código em uma aplicação.

***Refactorings* associados ao *bad smells*:** *Migração de Trecho de Código para um Novo Método* (usado para composição de métodos), *Migração de Trecho de Código para uma Nova Classe* (usado para mover funcionalidades entre objetos), *Migração de Método para Superclasse* e *Migração de Trecho de Código para um Novo Método Modelo* (usados para trabalhar com generalizações).

Possibilidades mais prováveis de ocorrência do *bad smells*: conforme descrito na Tabela 4.1.

Tabela 4.1: Possibilidades mais prováveis de ocorrência do *bad smells* Código Duplicado e suas respectivas sugestões

Possibilidades mais prováveis de ocorrência do *Bad Smell*	**Sugestão de correção do *Bad Smell* através de *refactoring***
A mesma expressão existe em dois métodos da mesma classe.	*Migração de Trecho de Código para um Novo Método*
A mesma expressão existe em duas suclasses da mesma classe.	*Migração de Trecho de Código para um Novo Método* e *Migração de Variável para Superclasse*, respectivamente.
Duas ou mais ocorrências de código similar	*Migração de Trecho de Código para um Novo Método* e *Migração de Trecho de Código para um Novo Método Modelo*, respectivamente.
Os métodos possuem o mesmo objetivo com algoritimos diferentes.	Escolher o melhor dos dois algoritimos e depois usar o *refactoring Substituição de Algoritmo*.
O código duplicado ocorre em duas classes não relacionadas.	Aplicar *Migração de Trecho de Código para uma Nova Classe* em uma das classes e depois usar o novo componente na outra classe.

Aplicando a abordagem MPM no *bad smells*: Analisar o código fonte com o objetivo de melhorá-lo (*refactoring*) com respeito a Código Duplicado do ponto de vista do programador e da equipe de manutenção.

Como resultado da aplicação da abordagem MPM no *bad smells* Código Duplicado, tem –se a Tabela 4.2.

Tabela 4.2. Aplicando a abordagem MPM com a meta de detectar o *bad smells* Código Duplicado

Pergunta	Métricas
1) A mesma expressão existe em dois métodos da mesma classe?	**M1.1:** Podem ser obtidas através de análise estática qualitativa. Não difundidas na literatura (Tipo3).
2) Há ocorrências do mesmo código em duas suclasses da mesma classe?	**M1.2:** Podem ser obtidas através de análise estática qualitativa. Não difundidas na literatura (Tipo3).
3) O código é similar, mas não é o mesmo?	**M1.3:** Resultado fortemente dependente de análise cognitiva, sendo, portanto, de difícil implementação. Não difundidas na literatura (Tipo5).
4) Os métodos têm o mesmo objetivo com algoritmos diferentes?	**M1.4:** Através de análise estática qualitativa podem ser obtidas métricas auxiliar no processo de obtenção do resultado desejado. Não difundidas na literatura (Tipo3).

4.1.2.2. Método Longo

Segundo Fowler (2000a), aplicações com métodos curtos são de melhor compreensão para o programador e para a equipe de manutenção. Deve-se sempre evitar métodos longos. Sempre que houver a necessidade de adicionar um comentário, deve-se acrescentar um novo método. Este método terá o conteúdo anteriormente comentado, sendo seu nome de fundamental importância para facilitar a compreensão do método em questão. Isto pode ser aplicado desde em um grupo de linhas de código até em uma única linha, mesmo na situação quando o nome do método e sua assinatura forem maiores que o trecho do código recém modificado. A questão principal é a distância semântica existente entre o que o método faz e como ele o faz. Uma parte do código comentado que indica o que está sendo feito pode ser substituído por um método cujo nome pode traduzir ou indicar aquilo que estava no comentário.

***Refactorings* associados ao *bad smells*:** *Migração de Trecho de Código para um Novo Método*, *Substituir Variável Temporaria por Consulta*, *Substituição de Método por Objeto* (usados para composição de métodos) *Substituição de Parâmetro por*

Objeto, *Uso de Objeto para Obtenção de Informações* (usados para tornar a chamada de métodos mais simples), e *Decomposição de Condicional* (usado para simplificar expressões condicionais).

Possibilidades mais prováveis de ocorrência do *bad smells*: conforme descrito na Tabela 4.3.

Tabela 4.3: Possibilidades mais prováveis de ocorrência do *bad smells* Método Longo e suas respectivas sugestões

Possibilidades mais prováveis de ocorrência do *Bad Smell*	**Sugestão de correção do *Bad Smell* através de *refactoring***
O tamanho dos métodos está acima do necessário.	*Migração de Trecho de Código para um Novo Método*
Existem métodos com lista longa de parâmetros.	*Substituição de Parâmetro por Objeto* e *Uso de Objeto para Obtenção de Informações*, respectivamente. Ou *Substituição de Método por Objeto*.
Existem métodos com muitas variáveis temporárias.	*Substituir Variável Temporaria por Consulta*
Existem métodos com estruturas condicionais e laços em excesso.	*Decomposição de Condicional*

Aplicando a abordagem MPM no *bad smells*: Analisar o código fonte com o objetivo de melhorá-lo (*refactoring*) com respeito a Método Longo do ponto de vista do programador e da equipe de manutenção.

Resultados obtidos: Como resultado da aplicação da abordagem MPM no *bad smells* Método Longo, tem –se a Tabela 4.4.

Tabela 4.4. Aplicando a abordagem MPM com a meta de detectar o *bad smells* Método Longo

Pergunta	Métricas
1) Qual o tamanho do método?	**M2.1:** *Number of Statements (NOS).* Há a possibilidade de derivação de outras métricas (Tipo2): a) tamanho do método em relação ao tamanho da classe; b) tamanho do método em relação ao tamanho médio dos métodos que compõem a classe.
2) Existem comentários que podem ser substituídos por métodos?	**M2.2:** Resultado fortemente dependente de análise cognitiva, sendo, portanto, de difícil implementação. Não difundidas na literatura (Tipo5).
3) O método tem parâmetros e variáveis temporárias que podem ser eliminados?	**M2.3:** As métricas do tipo Número de Variáveis temporárias e Números de parâmetros podem auxiliar no processo de análise cognitiva para a obtenção do resultado desejado (Tipo1).
4) Estruturas condicionais e laços podem ser eliminados ou simplificados?	**M2.4:** *Cyclomatic Complexity* pode auxiliar no processo de análise cognitiva para a obtenção do resultado desejado (Tipo1).

4.1.2.3. Classe Longa

A solução mais simples para reduzir o tamanho de uma classe considerada longa é eliminando-se a redundância. Neste caso, também podem ser seguidas as orientações fornecidas no item Método Longo. Para os outros casos, devem ser seguidas as orientações abaixo para que haja a possibilidade de segmentação do código da classe analisada.

***Refactorings* associados ao *bad smells*:** *Migração de Trecho de Código para uma Nova Classe*, *Migração de Trecho de Código para uma Nova Subclasse, Migração de Trecho de Código para uma Nova Interface, Separação da Interface da Lógica do*

Negócio. Sendo os três primeiros classificados como *refactorings* para trabalhar com generalizações e o último para organização de dados.

Possibilidades mais prováveis de ocorrência do *bad smells*: conforme descrito na Tabela 4.5.

Tabela 4.5: Possibilidades mais prováveis de ocorrência do *bad smells* Classe Longa e suas respectivas sugestões

Possibilidades mais prováveis de ocorrência do *Bad Smell*	**Sugestão de correção do *Bad Smell* através de *refactoring***
Classes longas com muitas variáveis de instância.	*Migração de Trecho de Código para uma Nova Classe* para agrupar e reduzir estas variáveis.
Sufixos e Prefixos comuns para um conjunto de variáveis em uma classe sugerem a oportunidade de criação de um componente.	*Migração de Trecho de Código para uma Nova Subclasse* para a criação de uma subclasse.
Determinado grupo de classes utiliza somente parte dos métodos disponibilizados por uma classe específica.	*Migração de Trecho de Código para uma Nova Interface* para colocar na nova interface a assinatura de todos os métodos solicitado para o grupo de classes em questão.
A classe estudada refere-se a uma Interface Gráfica de Usuário contendo também código associado a dados e comportamento.	Mover os dados e o comportamento para duas classes distintas. Como conseqüência, ter-se-á a necessidade de manutenção de dados duplicados em dois lugares e de forma sincronizada através de *Separação da Interface da Lógica do Negócio*.

Aplicando a abordagem MPM no *bad smells*: Analisar o código fonte com o objetivo de melhorá-lo (*refactoring*) com respeito a Classe Longa do ponto de vista do programador e da equipe de manutenção.

Tabela 4.6. Aplicando a abordagem MPM com a meta de detectar o *bad smells* Classe Longa

Pergunta	Métricas
1) Qual o tamanho da classe?	**M3.1:** Lines of Code (LOC). Há a possibilidade de uso de outras métricas (Tipo 2): a) tamanho da classe em relação ao tamaho da aplicação (soma de todas as classes e interfaces); b) tamanho da classe em relação ao tamanho médio das classes que compõem a aplicação.
2) Existem variáveis de instância e métodos que podem ser agrupados em uma nova classe criada para tal fim?	**M3.2:** (Tipo 1) *Number of Members* (NOM); *Response for Class* (RFC), *Lack Of Cohesion Of Methods 1* (LOCOM1), *Lack Of Cohesion Of Methods 2*(LOCOM2), *Lack Of Cohesion Of Methods 3* (LOCOM3).
3) Existem variáveis de instância que podem ser eliminadas?	**M3.3:** (Tipo 1) Número de variáveis de instância.
4) O método tem parâmetros e variáveis temporárias que podem ser eliminados?	**M3.4:** (Tipo 1) Número de Variáveis temporárias e parâmetros e *Maximun Number of Parameters* (MNOP).
5) Existem grupos de classes que utilizam somente parte dos métodos disponibilizados por uma classe específica?	**M3.5:** (Tipo 1) *Coupling Between Objects* (CBO), *Method Invocation Coupling* (MIC) e *Number of Remote Methods* (NORM).

4.1.2.4. Lista Longa de Parâmetros

Caso seja necessária a obtenção de um determinado conjunto de informações é possível solicitá-lo a um objeto. Com o uso de objetos, não é necessário informar tudo que o método necessita, mas o suficiente para que o mesmo possa atingir o seu objetivo. Em programação orientada a objetos, a lista de parâmetros tende a ser menor que no paradigma procedural. Lista de parâmetros longas são de difícil compreensão e uso.

***Refactorings* associados ao *bad smells*:** *Substituição de Parâmetro por Método*, *Uso de Objeto para Obtenção de Informações, Substituição de Parâmetro por Objeto.* Todos pertencentes à classificação Tornando a Chamada de Métodos mais Simples apresentada na Tabela 4.7.

Tabela 4.7: Possibilidades mais prováveis de ocorrência do *bad smells* Lista Longa de Parâmetros e suas respectivas sugestões

Possibilidades mais prováveis de ocorrência do *Bad Smell*	**Sugestão de correção do *Bad Smell* através de *refactoring***
Um objeto chama um método cujo retorno é passado como parâmetro para um outro método.	Dados podem ser obtidos através da requisição a um objeto que se tem acesso. Use *Substituição de Parâmetro por Método* para remover o parâmetro e permitir que o método seja chamado diretamente pelo objeto interessado.
Dois ou mais valores estão sendo obtidos de um objeto e repassados como parâmetros através da chamada de um método.	Passe o objeto no lugar dos parâmetros através de *Uso de Objeto para Obtenção de Informações.*
Um conjunto de parâmetros com possibilidade de ser agrupado, pois muitos métodos tendem a utilizá-los de forma agrupada.	Substitua-os através de *Substituição de Parâmetro por Objeto*.

Aplicando a abordagem MPM no *bad smells*: Analisar o código fonte com o objetivo de melhorá-lo (*refactoring*) com respeito a Lista Longa de Parâmetros do ponto de vista do programador e da equipe de manutenção.

Tabela 4.8: Aplicando a abordagem MPM com a meta de detectar o *bad smells* Lista Longa de Parâmetros

Pergunta	Métricas
1) Existem métodos cujo retorno são passados como parâmetros para outros métodos?	**M4.1:** (Tipo 3) Possibilidade de uso de análise estática qualitativa. Não difundidas na literatura.
2) Dois ou mais valores estão sendo obtidos de um objeto e repassados como parâmetros através da chamada de um método?	**M4.2:** (Tipo 3) Possibilidade de uso de análise estática qualitativa.Não difundidas na literatura.
3) Existem conjuntos de parâmetros com possibilidade de serem agrupados (levando-se em consideração que dois ou mais métodos tendem a utilizá-los de forma agrupada)?	**M4.3:** (Tipo 3) Possibilidade de uso de análise estática qualitativa.Não difundidas na literatura.

4.1.2.5. Mudanças Divergentes

O software deve ser estruturado de tal forma que mudanças possam ser realizadas facilmente. Mudanças divergentes ocorrem quando uma classe pode vir a ser modificada de diferentes formas por diferentes razões. Assim, para uma determinada necessidade, tem-se a modificação de um determinado número de classes, enquanto que para outra necessidade, tem-se um número diferente de classes a ser modificada. Neste caso, duas classes são melhores do que uma, não permitindo a existência de mudanças divergentes e permitindo que cada classe venha a ser modificada em função de uma única causa de mudança.

***Refactorings* associados ao *bad smells*:** *Migração de Trecho de Código para uma Nova Classe* (usado para composição de métodos).

Tabela 4.9: Possibilidades mais prováveis de ocorrência do *bad smells* Mudanças Divergentes e suas respectivas sugestões

Possibilidades mais prováveis de ocorrência do *Bad Smell*	**Sugestão de correção do *Bad Smell* através de *refactoring***
Classes podem vir a sofrer diferentes tipos de modificações em função de diferentes necessidades.	Identificação das mudanças associadas a cada causa específica para que seja possível agrupar todos os componentes envolvidos com esta causa específica em uma nova classe através do *refactoring* *Migração de Trecho de Código para uma Nova Classe*.

Aplicando a abordagem MPM no *bad smells*: Analisar o código fonte com o objetivo de melhorá-lo (*refactoring*) com respeito a Mudanças divergentes do ponto de vista do programador e da equipe de manutenção.

Tabela 4.10: Aplicando a abordagem MPM com a meta de detectar o *bad smells* Mudanças Divergentes

Pergunta	Métricas
1)Quais as causas possíveis de mudança para uma determinada classe ?	**M5.1:** (Tipo 5) Resultado fortemente dependente de análise cognitiva, sendo, portanto, de difícil implementação. Não difundidas na literatura.
2) Para as causas identificadas através das métricas do tipo M5.1, quais os componentes de uma classe (métodos e variáveis) que podem ser modificados em função das possíveis causas?	**M5.2**: (Tipo4) Conjunto de métricas derivada das métricas do tipo M5.1. Não difundidas na literatura.
3) Um mesmo componente pode ser modificado devido a diferentes causas de mudança identificadas através das métricas do tipo M5.1?	**M5.3**: (Tipo 4) Conjunto de métricas derivada das métricas do tipo M5.1. Não difundidas na literatura.

4.1.2.6. Mudanças em Cascata

É o oposto de mudanças divergentes. Percebe-se a sua ocorrência toda vez que para se promover algum tipo de alteração, tem-se a necessidade de se realizar várias pequenas modificações em diversas classes diferentes. Mudanças divergentes ocorrem quando uma classe é atingida por vários tipos de mudanças, ao passo que em mudanças em cascata uma alteração ocasiona modificações em diversas classes.

***Refactorings* associados ao *bad smells*:** *Migração de Método para Outra Classe*, *Migração de Variável para Outra Classe* e *Dissolução de Classe* (usados para mover funcionalidades entre objetos).

Tabela 4.11: Possibilidades mais prováveis de ocorrência do *bad smells* Mudanças em Cascata e suas respectivas sugestões

Possibilidades mais prováveis de ocorrência do *bad smells*	**Sugestão de correção do *bad smells* através de *refactoring***
Situações onde as modificações efetuadas estejam espalhadas pelas diversas classes que compõem a aplicação, não sendo de fácil localização, além de ser possível desconsiderar alguma modificação feita em alguma classe.	Uso de *Migração de Método para Outra Classe* e *Migração de Variável para Outra Classe* para se colocar todas as modificações em uma única classe.
Comportamentos relacionados estão presentes em classes diferentes.	Reunião dos comportamentos relacionados em uma única classe através de *Dissolução de Classe*.

Aplicando a abordagem MPM, tem-se: Analisar o código fonte com o objetivo de melhorá-lo (*refactoring*) com respeito a Mudanças em Cascata do ponto de vista do programador e da equipe de manutenção.

Tabela 4.12: Aplicando a abordagem MPM com a meta de detectar o *bad smells* Mudanças em Cascata

Pergunta	Métricas
1) Ao se promover algum tipo de alteração, tem-se a necessidade de se realizar várias pequenas modificações em diversas classes diferentes?	**M6.1:** (Tipo 4) Conjunto de métricas derivada das métricas do tipo M5.1. Não difundidas na literatura.

4.1.2.7. Obtenção de informações de objetos instanciados a partir de outras classes

Uma das características principais dos objetos é o encapsulamento. Um *bad smells* clássico ocorre quando um método solicita mais informações de objetos instanciados a partir de outras classes do que da classe a qual pertence. O foco principal deste interesse são os dados.

Alguns padrões podem ser usados para resolver esta situação: *Strategy* e *Visitor* propostos pela *Gang* dos Quatro - alusão geralmente feita aos autores do livro de referência (Gamma e outros, 1995) em reconhecimento ao importante trabalho realizado pelos mesmos na área de padrões de projeto - ou *Self Delegation* proposto por Beck (1999). A principal regra a ser seguida é colocar os componentes afins juntos.

***Refactorings* associados ao *bad smell*:** *Migração de Trecho de Código para um Novo Método* (usado para composição de métodos)*, Migração de Método para Outra Classe* e *Dissolução de Classe* (usados para mover funcionalidades entre objetos)

Tabela 4.13: Possibilidades mais prováveis de ocorrência do *bad smells* Obtenção de informações de objetos instanciados a partir de outras classes e suas respectivas sugestões

Possibilidades mais prováveis de ocorrência do *Bad Smell*	**Sugestão de correção do *Bad Smell* através de *Refactoring***
Um determinado método invoca com frequência métodos de outros objetos para obter informações necessárias para alguma atividade.	A solução mais aparente é o uso do refactoring *Migração de Método para Outra Classe* .
Apenas uma parte do método tem a característica descrita acima.	Uso do *Migração de Trecho de Código para um Novo Método* na parte do código que tem a característica descrita anteriomente para que, em seguida, seja aplicado *Migração de Método para Outra Classe* no novo método criado.
Um método usa recursos de várias classes.	Deve-se considerar a classe que tem o maior volume de informações a ser oferecida ao método em estudo e colocar o método na mesma. Este processo pode se tornar mais simples com o uso de *Migração de Trecho de Código para um Novo Método* para dividir o método e deslocar o novo método criado para a classe indicada.

Aplicando a abordagem MPM no *bad smells*: Analisar o código fonte com o objetivo de melhorá-lo (refactoring) com respeito a Obtenção de informações de objetos instanciados a partir de outras classes do ponto de vista do programador e da equipe de manutenção.

Tabela 4.14: Aplicando a abordagem MPM com a meta de detectar o *bad smells* Obtenção de Informações de Objetos Instanciados a partir de Outras Classes.

Pergunta	Métricas
1) Qual o número de vezes que os métodos solicitam informações de objetos instanciados a partir de outras classes?	**M7.1:** (Tipo 1) *Number of Remote Methods* (NORM).
2) Dos métodos obtidos através das métricas M7.1 em quais deles apenas uma parte tem a característica descrita acima?	**M7.2:** (Tipo 4) Conjunto de métricas derivada das métricas do tipo M7.1. Não difundidas na literatura.

4.1.2.8. Grupos de Dados

Grupos de Dados afins devem ser posicionados em objetos comuns.

***Refactorings* associados ao *bad smells*:** *Migração de Trecho de Código para uma Nova Classe* (usado para mover funcionalidades entre objetos), *Substituição de Parâmetro por Objeto* e *Uso de Objeto para Obtenção de Informações* (usados para tornar a chamada aos métodos mais simples).

Tabela 4.15: Possibilidades mais prováveis de ocorrência do *bad smells* Grupos de Dados e suas respectivas sugestões

Possibilidades mais prováveis de ocorrência do *Bad Smell*	**Sugestão de correção do *Bad Smell* através de *Refactoring***
Grupos de dados aparecem como atributos.	Uso de *Migração de Trecho de Código para uma Nova Classe* nestes atributos para torná-los pertencentes a uma nova classe específica para tal fim.
Há a possibilidade de simplificação da lista de parâmetros que compõem a assinatura dos métodos através da sua substituição por objetos.	Uso dos *refactorings Substituição de Parâmetro por Objeto* ou *Uso de Objeto para Obtenção de Informações*. O benefício imediato é a possibilidade de simplificar a lista de parâmetros e a chamado aos métodos.
Determinado grupo de parâmetros sempre é utilizado quando da chamada de diferentes métodos.	Uso de *Substituição de Parâmetro por Objeto* para promover o encapsulamento deste grupo de parâmetros em um objeto.
Estão sendo passados como parâmetros valores obtidos de um ou mais objetos.	Uso do *refactoring Uso de Objeto para Obtenção de Informações* para que seja possível o uso do(s) próprio(s) objeto(s) como parâmetro(s).

Aplicando a abordagem MPM no *bad smells*: Analisar o código fonte com o objetivo de melhorá-lo (*refactoring*) com respeito a Grupos de Dados do ponto de vista do programador e da equipe de manutenção.

Tabela 4.16: Aplicando a abordagem MPM com a meta de detectar o *bad smells* Grupos de Dados

Pergunta	Métrica
1) Existem conjuntos de dados que podem ser agrupados em um único objeto?	**M8.1**: (Tipo 5) Resultado dependente fortemente de análise cognitiva, sendo, portanto, de difícil implementação. Não difundidas na literatura

4.1.2.9. Resistência na utilização de Objetos para pequenas atividades

Para aqueles não habituados ao uso de objetos, percebe-se certa resistência no uso dos mesmos para pequenas atividades: classes que combinam número e moeda, faixas de valores com limites superiores e inferiores e seqüência de caracteres tais como números de telefone e CEP, entre outros. Todos estes exemplos podem ser trabalhados na forma de objetos.

***Refactorings* associados ao *bad smells*:** *Criação de Objeto para Tratamento de Informações*, *Substituição de Tipo de Código por Classe*, *Substituição de Tipo de Código por Subclasse*, *Substituição de Tipo de Código por Objeto do Tipo Estado/Estratégia* (todos usados para organização de dados).

Aplicando a abordagem MPM no *bad smells*: Analisar o código fonte com o objetivo de melhorá-lo (*refactoring*) com respeito a Resistência na utilização de Objetos para pequenas atividades do ponto de vista do programador e da equipe de manutenção.

Tabela 4.17: Possibilidades mais prováveis de ocorrência do *bad smells* Resistência na utilização de Objetos para pequenas atividades e suas respectivas sugestões

Possibilidades mais prováveis de ocorrência do *Bad Smell*	**Sugestão de correção do *Bad Smell* através de *Refactoring***
Verifica-se que um determinado atributo necessita de mais de uma forma de representação, formatação, extração de informações que o compõem (área de código de um número de telefone, prefixo, entre outros).	*Criação de Objeto para Tratamento de Informações* para valores de dados individuais.
Um classe tem tipo de código numérico que não afeta o seu comportamento.	*Substituição de Tipo de Código por Classe* para substituir o atributo numérico por uma nova classe.
Existem tipos de código imutáveis que afetam o comportamento da classe.	Substitua o tipo de código por uma subclasse usando *Substituição de Tipo de Código por Subclasse.*
Existem *type codes* imutáveis que afetam o comportamento da classe, mas que não podem ser substituídos por subclasses.	Substitua o tipo de código por um objeto do tipo estado usando *Substituição de Tipo de Código por Objeto do Tipo Estado/Estratégia.*

Tabela 4.18: Aplicando a Metodologia MPM com a meta de detectar o *bad smells* Resistência de utilização de objetos para pequenas atividades

Pergunta	**Métricas**
1) Existem informações que podem ser mantidas em uma classe específica para o uso das mesmas?	**M9.1:** (Tipo 5) Resultado fortemente dependente de análise cognitiva, sendo, portanto, de difícil implementação. Não difundidas na literatura.
2) Existem nas classes *type code* numéricos que não afetam o comportamento e que podem ser mantidas em uma classe específica para o uso das mesmas?	**M9.2**: (Tipo 3) Possibilidade de uso de análise estática qualitativa.Não difundidas na literatura.
3) Um determinado atributo necessita de mais de uma forma de representação, formatação ou extração de informações que o compõem (área de código de um número de telefone, prefixo, entre outros).	**M9.3:** (Tipo 5) Resultado fortemente dependente de análise cognitiva, sendo, portanto, de difícil implementação. Não difundidas na literatura.

4.1.2.10.Sentenças com *Switch*

Um dos problemas do uso de sentenças com *switch* em orientação a objetos é a duplicação. Geralmente encontramos sentenças com *switch* espalhadas pelo código. Ao se adicionar uma nova cláusula ao *switch*, poderá ocorrer a necessidade de pesquisa em todas as sentenças com *switch* para a modificação das mesmas. O conceito de polimorfismo em orientação a objetos proporciona uma forma elegante de se resolver este problema. Na maioria das vezes que ocorre uma sentença com *switch*, poderá ser aplicado o conceito de polimorfismo. Geralmente a sentença com *switch* ocorre com *type code*. Deseja-se um método ou classe que contenha o valor do *type code* para posterior eliminação das sentenças com *switch*.

***Refactorings* associados ao *bad smells*:** *Migração de Trecho de Código para um Novo Método* (usado para composição de métodos), *Migração de Método para Outra Classe* (usado para mover funcionalidades entre objetos)**,** *Substituição de Tipo de Código por Subclasses* e *Substituição de Tipo de Código por Objeto do Tipo Estado/Estratégia* (usados para organização de dados), *Substituição de Condicional por Polimorfismo e Uso de Objeto Nulo* (usados para simplificar expressões condicionais), *Substituição de Parâmetros por Métodos* (usados para tornar as chamadas aos métodos mais simples).

Tabela 4.19: Possibilidades mais prováveis de ocorrência do *bad smells* Sentenças com *Switch* e suas respectivas sugestões

Possibilidades mais prováveis de ocorrência do *Bad Smell*	**Sugestão de correção do *Bad Smell* através de *Refactoring***
Existem sentenças com switch numa determinada classe com possibilidades de serem conduzidas para novos métodos.	*Use Migração de Trecho de Código para um Novo Método para retirar o trecho com switch e Migração de Método para Outra Classe para mover o novo método para a classe onde será necessário usar o polimorfismo. Neste momento, será necessário decidir pelo uso de Substituição de Tipo de Código por Subclasses ou Substituição de Tipo de Código por Objeto do Tipo Estado/Estratégia.*
Há a necessidade de configuração da nova estrutura de herança.	*Uso de Substituição de Condicional por Polimorfismo.*
Existem poucas sentenças com *case* que afetam um único método, sendo que tais sentenças não sofrerão modificações.	Uso de *Substituição de Parâmetros por Métodos*.
Uma das condicionais case é nula.	*Uso do Uso de Objeto Nulo.*

Aplicando a abordagem MPM no *bad smells*: Analisar o código fonte com o objetivo de melhorá-lo (refactoring) com respeito a Sentenças com *Switch* do ponto de vista do programador e da equipe de manutenção.

Tabela 4.20: Aplicando a Metodologia MPM com a Meta de detectar o *bad smells* Sentenças com *Switch*

Pergunta	Métricas
1) Existem sentenças com *Switch* que podem ser substituídas por polimorfismo?	**M10.1:** (Tipo 3) Possibilidade de uso de análise estática qualitativa para tal finalidade. Não difundidas na literatura.

4.1.2.11. Hierarquia de Heranças Paralelas

Hierarquia de heranças paralelas é um caso especial do *bad smells* Mudanças em Cascata: toda vez que se criar uma subclasse, deverá também ser criada uma subclasse correspondente associada a outra classe.

***Refactorings* associados ao *bad smells*:** *Migração de Método para Outra Classe* e *Migração de Variável para Outra Classe* (usados para mover funcionalidades entre objetos).

Tabela 4.21: Possibilidades mais prováveis de ocorrência do *bad smells* Hierarquia de Heranças Paralelas e suas respectivas sugestões

Possibilidades mais prováveis de ocorrência do *Bad Smells*	**Sugestão de correção do *Bad Smells* através de *Refactoring***
Considerando a estrutura de hierarquia das classes que compõem a aplicação, existe a possibilidade de, na criação de uma subclasse, haver o comprometimento de criação também de uma subclasse associada a uma outra classe.	A estratégia para a eliminar este problema é detectar instâncias de uma hierarquia que fazem referência a instâncias de outra hierarquia em paralelo. Este problema pode ser detectado pelo reconhecimento de prefixos dos nomes das classes em uma hierarquia coincidindo com os prefixos de outra hierarquia. Pode-se usar *Migração de Método para Outra Classe* e *Migração de Variável para Outra Classe* para o desaparecimento deste problema.

Aplicando a abordagem MPM no *bad smells*: Analisar o código fonte com o objetivo de melhorá-lo (refactoring) com respeito a Hierarquia de herança paralelas do ponto de vista do programador e da equipe de manutenção.

Tabela 4.22: Aplicando a abordagem MPM com a meta de detectar o *bad smells* Hierarquia de Heranças Paralelas

Pergunta	Métricas
1) Considerando a estrutura de hierarquia das classes que compõem a aplicação, existe a possibilidade de, na criação de uma subclasse, haver o comprometimento de criação também de uma subclasse associada a uma outra classe?	**M11.1**: (Tipo 6) Possibilidade de uso de análise dinâmica para tal finalidade. Não difundidas na literatura.
2) Os Prefixos dos nomes das classes em uma hierarquia de herança de classes coincidem com os prefixos de outra hierarquia?	**M11.2**: (Tipo 3) Possibilidade de uso de análise estática qualitativa. Não difundidas na literatura

4.1.2.12. Classes Supérfluas

Para cada classe criada tem-se um custo associado a sua manutenção e compreensão por parte da equipe de desenvolvimento e ou manutenção. Para este caso, o mais importante é verificação de classes cuja existência não justifiquem o seu custo para posterior exclusão.

***Refactorings* associados ao *bad smells*:** *Junção de Superclasse e Subclasse* (usados para trabalhar com generalizações*)* e *Dissolução de Classe* (usado para mover funcionalidades entre objetos).

Tabela 4.23: Possibilidades mais prováveis de ocorrência do *bad smells* Classes Supérfluas e suas respectivas sugestões

Possibilidades mais prováveis de ocorrência do *Bad Smell*	**Sugestão de correção do *Bad Smell* através de *Refactoring***
Existem classes que foram criadas como conseqüência de algum *refactoring* ou foram adicionadas devido a mudanças que foram planejadas, mas não realizadas.	Nos casos de subclasses que não estando sendo necessárias pode-se usar *Junção de Superclasse e Subclasse* . Nos demais casos pode-se usar *Dissolução de Classe.*

Aplicando a abordagem MPM no *bad smells*: Analisar o código fonte com o objetivo de melhorá-lo (refactoring) com respeito a Classes Supérfluas do ponto de vista do programador e da equipe de manutenção.

Tabela 4.24: Aplicando a Metodologia MPM com a Meta de detectar o *bad smells* Classes Supérfluas

Pergunta	Métrica
1) Existem classes não utilizadas ou cuja existência não é justificada?	**M12.1:** (Tipo 6) Possibilidade de uso de análise dinâmica para tal finalidade. Não difundidas na literatura.

4.1.2.13. Recursos Desnecessários

É comum a introdução de recursos para uso futuro que, na realidade, não são utilizados e passam a não ter a sua existência justificada. A presença de recursos desnecessários pode ser sinalizado sempre que os únicos usuários de um método ou de uma classe forem os mecanismos de realização de testes. Caso seja encontrada tal situação, deve-se removê-los juntamente com os itens de teste correspondentes.

***Refactorings* associados ao *bad smells*:** *Junção de Superclasse e Subclasse* (usado para trabalhar com generalizações), *Dissolução de Classe* (usado para mover funcionalidades entre objetos), *Remoção de Parâmetros* e *Renomeando Métodos* (usados para tornar a chamada aos métodos mais simples).

Tabela 4.25: Possibilidades mais prováveis de ocorrência do *bad smells* Recursos Desnecessários e suas respectivas sugestões

Possibilidades mais prováveis de ocorrência do *Bad Smell*	**Sugestão de correção do *Bad Smell* através de *Refactoring***
Classes abstratas não estão sendo usadas.	Uso de *Junção de Superclasse e Subclasse.*
Existem delegações desnecessárias.	Uso de *Dissolução de Classe.*
Existem métodos com parâmetros sem uso.	Uso de *Remoção de Parâmetros*.
Existem métodos cujos nomes podem ser modificados.	Uso de *Renomeando Métodos*.

Aplicando a abordagem MPM no *bad smells*: Analisar o código fonte com o objetivo de melhorá-lo (refactoring) com respeito a Recursos Desnecessários do ponto de vista do programador e da equipe de manutenção.

Tabela 4.26: Aplicando a Metodologia MPM com a Meta de detectar o *bad smells* Recursos Desnecessários

Pergunta	Métricas
1) Classes abstratas não estão sendo usadas ou são desnecessárias?	**M13.1:** (Tipo 3) Possibilidade de uso de análise estática qualitativa para tal finalidade. Não difundidas na literatura.
2) Existem classes desnecessárias que não estão justificando a sua existência?	**M13.2:** (Tipo 6) Possibilidade de uso de análise dinâmica para tal finalidade. Não difundidas na literatura. Mesmo conjunto de métricas identificado como M12.1.
3) Existem métodos com parâmetros sem uso?	**M13.3:** (Tipo 3) Possibilidade de uso de análise estática qualitativa para tal finalidade. Não difundidas na literatura.

4.1.2.14. Campo Temporário

É comum um objeto ter uma variável de instância sendo utilizada somente em raras circunstâncias. A tentativa de descobrir o motivo da existência de uma variável pode ser uma tarefa que consuma tempo.

***Refactorings* associados ao *bad smells*:** *Migração de Trecho de Código para uma Nova Classe* (usado para mover funcionalidades entre objetos).

Tabela 4.27: Possibilidades mais prováveis de ocorrência do *bad smells* Campo Temporário e suas respectivas sugestões

Possibilidades mais prováveis de ocorrência do *Bad Smell*	**Sugestão de correção do *Bad Smell* através de *Refactoring***
Existe um conjunto de variáveis de instância em um objeto sendo utilizada somente em raras ou nenhuma circunstância.	Uso de *Migração de Trecho de Código para uma Nova Classe* para a criação de um local comum para estas variáveis.

Aplicando a abordagem MPM no *bad smells*: Analisar o código fonte com o objetivo de melhorá-lo (refactoring) com respeito a Campo Temporário do ponto de vista do programador e da equipe de manutenção.

Tabela 4.28: Aplicando a abordagem MPM com a meta de detectar o *bad smells* Campo Temporário

Pergunta	Métricas
1)Qual a frequência de uso das variáveis de instância?	**M14.1:** (Tipo 6) Possibilidade de uso de análise dinâmica para tal finalidade. Não difundidas na literatura.

4.1.2.15.Mensagens Encadeadas

Verifica-se a existência de mensagens encadeadas quando uma única mensagem recebido por um objeto faz com seja disparada uma mensagem para outros objetos de forma encadeada e sucessiva.

***Refactorings* associados ao *bad smells*:** *Migração de Método para Outra Classe* (usado para mover funcionalidades entre objetos), *Migração de Trecho de Código para um Novo Método* (usado para composição de métodos).

Tabela 4.29: Possibilidades mais prováveis de ocorrência do *bad smells* Mensagens Encadeadas e suas respectivas sugestões

Possibilidades mais prováveis de ocorrência do *Bad Smell*	**Sugestão de correção do *Bad Smell* através de *Refactoring***
Existem cadeias de mensagens disparadas a partir de um único objeto.	A melhor alternativa é verificar para que está sendo usado o objeto resultante. Neste caso, pode-se usar *Migração de Trecho de Código para um Novo Método* para retirar parte do código responsável pelo utilizado e conduzi-lo a um método específico e depois *Migração de Método para Outra Classe* para que o mesmo seja conduzido à parte inferior da cadeia.

Aplicando a abordagem MPM no *bad smells*: Analisar o código fonte com o objetivo de melhorá-lo (*refactoring*) com respeito a Mensagens Encadeadas do ponto de vista do programador e da equipe de manutenção.

Tabela 4.30: Aplicando a abordagem MPM com a meta de detectar o *bad smells* Mensagens Encadeadas

Pergunta	Métrica
1) Existem cadeias de mensagens disparadas a partir de um único objeto?	**M15.1**: (Tipo 6) Possibilidade de uso de análise dinâmica para tal finalidade. Não difundidas na literatura.

4.1.2.16. Intermediários Desnecessários

Sabe-se que uma das principais características de orientação a objetos é o encapsulamento: detalhes internos de implementação do objeto não são necessários para a obtenção de informações nele armazenadas, tais informações somente podem ser obtidas ou modificadas através de métodos que expressam o seu comportamento. Nestas circunstâncias, é possível a ocorrência de "delegação de atribuições": envia-se uma mensagem a um objeto para a execução de uma determinada atividade, sendo que este repassa a atividade para que um outro objeto a execute. O uso excessivo de delegação

de atribuições não é aconselhável. Quando constatado, pode-se avaliar a possibilidade de direcionar a solicitação diretamente ao objeto executor da atividade, excluindo-se os intermediários desnecessários.

***Refactorings* associados ao *bad smells*:** *Remoção de Delegação* (usado para mover funcionalidades entre objetos)*, Dissolução de Método* (usado para a composição de métodos) e *Substituição de Delegação por Herança* (usado para trabalhar com generalizações)

Tabela 4.31: Possibilidades mais prováveis de ocorrência do *bad smells* Intermediários Desnecessários e suas respectivas sugestões

Possibilidades mais prováveis de ocorrência do *Bad Smells*	**Sugestão de correção do *Bad Smells* através de *Refactoring***
Uma atividade é executada depois que a sua requisição foi repassada por uma cadeia considerada grande de objetos.	Uso de *Remoção de Delegação* para que a mensagem chegue diretamente ao objeto executor da atividade.

Aplicando a abordagem MPM no *bad smells*: Analisar o código fonte com o objetivo de melhorá-lo (*refactoring*) com respeito a Intermediários Desnecessários do ponto de vista do programador e da equipe de manutenção.

Tabela 4.32: Aplicando a abordagem MPM com a meta de detectar o *bad smells* Intermediários Desnecessários

Pergunta	Métricas
1) Uma atividade é executada depois que a sua requisição foi repassada por uma cadeia considerada grande de objetos.	**M16.1:** (Tipo 1) Existem métricas que podem contribuir para o diagnóstico: *Response for Class* (RFC), *Number of Remote Methods* (NORM), *Method Invocation Coupling* (MIC). Além disso, há a possibilidade de uso de análise estática qualitativa para derivação de métricas mais específicas para o fornecimento da reposta para a pergunta formulada.

4.1.3. Análise dos Resultados

Os resultados obtidos através do estudo de caso da abordagem *Top Down* em 16 dos *bad smells* contidos no catálogo apresentado por Fowler (2000a) indica que os 36 conjuntos de métricas foram distribuídos conforme apresentado na tabela 4.33. Dos 36 conjuntos de métricas, somente sete conjuntos são do tipo 1 (contém métricas disponíveis e indicadas para o auxílio do processo de análise cognitiva para obtenção) e dois conjuntos são do tipo 2 (conjuntos contém métricas disponíveis com possibilidade de derivação de novas métricas a partir do conjunto obtido). Os outros 27 conjuntos são métricas não disponíveis ou não difundidas distribuídas entre os demais tipos já apresentados na seção 4.1.1. Assim, a abordagem *Top Down* indica que há a necessidade de derivação de novas métricas que, utilizadas de forma combinada, sirvam para o auxílio nas etapas do uso de *refactoring* apresentadas na seção 2.3.

Tabela 4.33: Distribuição dos conjuntos de métricas obtidos da abordagem *Top Down* segundo os tipos previstos na seção 4.1.1.

Tipo de Métricas	**Conjuntos de métricas obtidos da abordagem *Top Down***	**Quantidade de conjuntos**	**Quantidade de conjuntos (%)**
1	M2.3, M2.4, ,M3.2, M3.3, M3.4, M3.5, M7.1	7	19,44
2	M2.1, M3.1	2	5,56
3	M1.1, M1.2, M1.4, M4.1, M4.2, ,M4.3, M9.2, M10.1, M11.2, M12.1, M13.1, M13.3	12	33,33
4	M5.2, M5.3, M6.1, M7.2	4	11,11
5	M1.3, M2.2, M5.1, ,M8.1, M9.1, M9.3	6	16,67
6	M11.1, M12.1, M13.2, M14.1, M15.1	5	13,89
Total		**36**	**100**

Pela tabela verifica-se que a porcentagem de métricas disponíveis (tipos 1 e 2) para suportar a metodologia é relativamente baixa: sendo 19,44% do tipo 1 e 5,56% do tipo 2, o que representa 25% da quantidade de conjuntos obtidos da abordagem *Top Down*.

Este é um dos principais obstáculos para a implementação da metodologia descrita na seção 3.4 (vide figura 3.7). Todavia, boa parte das métricas não disponíveis (58,33% distribuídos da seguinte forma: tipo 3 com 33,33%, tipo 4 com 11,11% e tipo 6 com 13,89%) são factíveis e implementáveis, enquanto que somente 16,67% das métricas não disponíveis são do tipo 5 (métricas não disponíveis ou não difundidas, cujo resultado depende fortemente de análise cognitiva, sendo, portanto, de difícil implementação). Este quadro indica que a metodologia descrita na figura 3.6 é factível, apesar das limitações inerentes à aplicação de métricas no auxílio do uso de *refactorings*, principalmente no que diz respeito à etapa de detecção.

4.2. ESTUDO DE CASO USANDO A ABORDAGEM *BOTTOM UP*

O objetivo deste estudo de caso é a coleta de informações a respeito do comportamento das métricas disponíveis antes, durante e depois do uso de uma seqüência específica de *refactorings* relacionadas para cada *bad smells* usando a abordagem *bottom up* (conforme já foi apresentado na seção 3.2).

4.2.1. Descrição do estudo de caso

O cenário usado é a aplicação de uma seqüência de *refactorings* em uma aplicação e obtenção de um conjunto de medidas antes, durante e depois da seqüência adotada.

Este cenário foi adotado levando-se em consideração que bons resultados no uso de *refactoring* somente serão obtidos se os mesmos forem utilizados de forma encadeada. O seu uso de forma isolada promove somente pequenas modificações, enquanto que sendo usado de forma encadeada e combinada tem grande impacto em uma aplicação (Fowler, 2000b).

Foi adotado no estudo de caso um exemplo obtido mediante consulta por correio eletrônico ao próprio Fowler. O exemplo demonstra uma seqüência de 77 *refactorings* agrupados em cinco *bad smells*. Foi considerado como aspecto principal o uso do conceito de herança, fazendo com que os *refactorings* proporcionassem um melhor uso deste conceito. A aplicação possui classes que inicialmente não estão relacionadas, mas que possuem comportamento similar. Ao longo do processo de *refactoring* tais classes são reestruturadas em uma superclasse e quatro subclasses, conforme pode ser

verificado através dos diagramas de classe apresentados no Apêndice 4. A aplicação utiliza a linguagem Java, sendo adotada pelas seguintes razões:

a) Representatividade da linguagem Java.

b) Amplo uso da linguagem em estudos sobre *refactoring*.

c) Acessibilidade: o código fonte é de acesso público (Fowler, 2000b), possibilitando que outros grupos de pesquisa possam reproduzir e expandir os resultados obtidos.

d) Representatividade: a aplicação já foi adotada em vários estudos de assuntos relacionados a *refactoring* por outros grupos de pesquisa (Fowler, 2000b).

e) Independência: a aplicação foi desenvolvida de forma independente do trabalho desta dissertação. As abordagens não influenciaram no processo de desenvolvimento do software.

f) Documentação: as características associadas a todas as fases desde a aplicação inicial, passando por cada *bad smells* e seus respectivos refactorings associados foram devidamente documentados em (Fowler, 2000b), possibilitando a validação dos resultados obtidos.

A principal limitação do estudo de caso é que os resultados obtidos estão condicionados aos *bad smells* detectados e à seqüência de *refactorings* adotada. O estudo de caso não aborda o conjunto completo de *bad smells* e *refactorings* sugeridos por Fowler em seu livro (Fowler, 2000a).

4.2.2. Artefatos utilizados

Neste estudo de caso foram consideradas como métricas disponíveis aquelas apresentadas na versão 6 do *Together Contol Center* (*TogetherControlCenter*, 2002). A relação destas métricas é apresentada no Apêndice 1.

4.2.3. Coleta de dados

Na coleta dos dados para análise foram consideradas:

a) Execução de todas as etapas de refactoring sugeridas na documentação. As etapas devem ser documentadas e organizadas separadamente para permitir a análise posterior do refactoring aplicado.

b) Obtenção das medidas disponibilizadas pela ferramenta Together Control Center versão 6.0 em cada etapa de refactoring.

c) Classificação das etapas segundo o refactoring aplicado e o bad smell ao qual está associado.

4.2.4. Processos de Análise

O objetivo da análise das medidas coletadas no estudo *bottom up* é fornecer subsídios para que nos resultados seja possível:

a) Verificar se o que foi previsto na abordagem *Top-Down* ocorreu.

b) Verificar ocorrências não previstas.

c) Identificar quais foram os itens previstos, mas que não ocorreram.

Na análise de dados verificou-se que cada *refactoring* tem sua área de impacto bem definida. Todo *refactoring* do tipo *Migração de Trecho de Código para um Novo Método*, por exemplo, modifica somente as medidas no escopo da classe, tendo em vista que o novo método criado, geralmente com modificador de acesso privado, será chamado a partir de um método antigo existente na própria classe.

Deverá ser considerado também que nem todas as influências causadas pelos *refactorings* foram capturadas, pois há a possibilidade do efeito anulador que uma ação tenha exercido sobre outra ação do mesmo *refactoring*, inibindo o resultado de variação para uma dada métrica.

4.2.4.1. Identificação das Principais Variações por Métrica

Para a análise dos dados obtidos no estudo de caso, segundo a abordagem *Bottom Up* (processo de "baixo nível"), foram adotadas as seguintes etapas conforme apresentado na figura 4.1:

Procedimento A: Medição dos valores para as "*métricas disponíveis*" em cada etapa de *refactoring* para cada classe. Estes valores são organizados em uma tabela.

Procedimento B: Obtenção da variação entre as etapas consecutivas de *refactoring* para cada métrica por classe. Descarte das métricas que não apresentaram variação.

Procedimento C: A partir da tabela obtida no procedimento B, separar os valores para cada métrica em tabelas distintas e acrescentar as informações de refactoring e bad smell associados a cada etapa.

Procedimento D: Para a obtenção das maiores variações para cada métrica foi adotada a técnica *Box plots* (Fenton, 1997). Para a aplicação desta técnica são obtidos os valores de mediana e os quatro quartis das métricas. Para que sejam analisados somente

os valores para uma dada métrica fortemente afetados, são considerados somente aqueles que não estiverem contidos na amplitude inter-quartil (medida que define a diferença entre os primeiro e o terceiro quartis).

Procedimento E: Cria-se para cada métrica disponível uma tabela para que sejam apresentados os *refactorings* que exerceram maior impacto na variação da mesma. Na tabela em questão, os atributos "*refactoring* afetado pela métrica" e "quantidade de valores fortemente afetados" são obtidos do *procedimento D*, "a quantidade de valores com variação" é obtida do *procedimento C* e o "total de *refactorings* aplicados no estudo de caso" é obtido após a aplicação de um filtro na tabela do *procedimento A,* para que sejam considerados somente os *refactorings* relacionados com a classe cujas medidas estão sendo analisadas.

A partir do *procedimento E* é obtida a informação de quais métricas apresentaram maior variação para um dado *refactoring*.

Por último, obtém-se um conjunto mínimo de métricas que são significativas para um dado *refactoring*. Para tanto, considerando como referência as métricas que apresentaram maior variação para um dado *refactoring*, são identificados conjuntos de métricas correlacionadas e escolhidas somente uma para cada grupo de métricas conceitualmente similares.

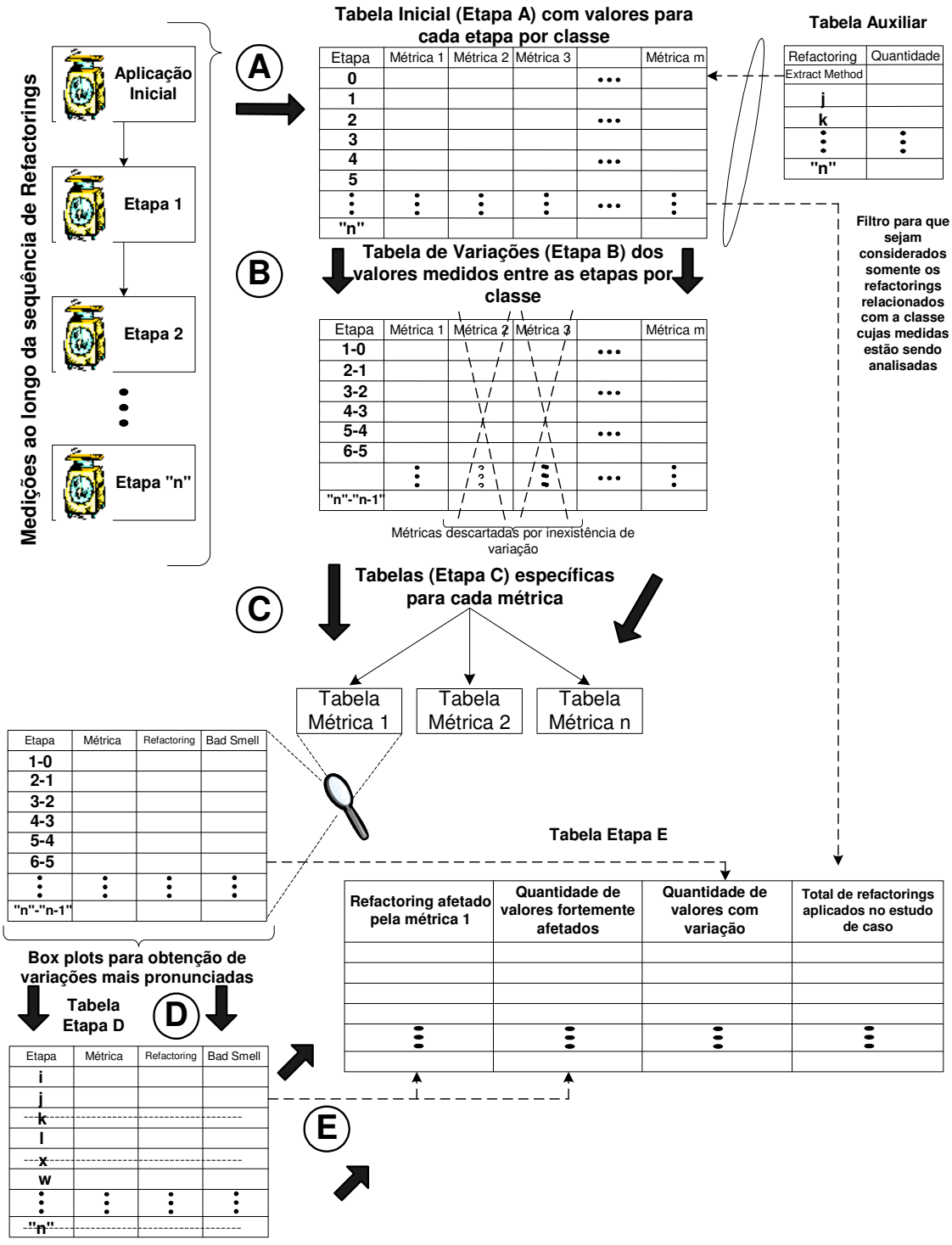


Figura 4.1: Processo de análise dos dados

4.2.4.2. Identificação das Métricas com maior Variação por *Refactoring*

Com os resultados obtidos do *procedimento E*, os refactorings similares foram agrupados para a posterior verificação das métricas afetadas e com maior variação.

4.2.4.3. Identificação das Métricas com maior Variação por *bad smells*

Espera-se também que os resultados obtidos do *procedimento E* sirvam para identificar quais métricas apresentaram maior variação antes e depois de cada *bad smells*.

4.2.5. Resultados do estudo de caso da abordagem *bottom up*

A seguir serão apresentados os resultados obtidos da análise das medidas da aplicação adotada para o estudo de caso usando a abordagem *bottom up*.

A existência de código duplicado possibilitou a criação de uma nova hierarquia de classes.

Seguindo o que foi descrito na seção anterior, serão consideradas as seguintes informações:

a) Número de classes afetadas: 8
b) Distribuição de influência de *refactorings* por classe conforme descrito na Tabela 4.34.

Tabela 4.34: Distribuição de ocorrências de *refactorings* nas classes que compõem a aplicação do estudo de caso

Classe	Ocorrências de *refactoring*	% Ocorrências de *refactoring* em relação ao Total
Site	26	29,55
DisabilitySite	19	21,59
ResidentialSite	17	19,32
LifelineSite	7	7,95
MfDate	7	7,95
BusinessSite	6	6,82
DateRange	4	4,55
Zone	2	2,27
Total	***88**	**100**

(*) O total de ocorrências de *refactorings* ultrapassou 77 pelo fato de alguns deles exercerem impacto em mais de uma classe simultaneamente.

A distribuição do número de ocorrências nas classes que compõem a aplicação retratam que à medida que são aplicados os *refactorings* na superclasse, tornou-se mais fácil promover a adaptação das demais com um menor número *refactorings*. Assim, verifica-se que a classe *Site* é a classe alvo já no primeiro *bad smell* com intensidade maior que nos demais conforme mostra a Tabela 4.35 a seguir.

Tabela 4.35: Distribuição de ocorrências de *refactorings* na classe *Site* ao longo dos *bad smells*

Bad Smell	Número de ocorrências na classe Site
1 – Código Duplicado	9
2 – Código Duplicado / Método Longo	6
3 – Método Longo	8
4 – Código Duplicado	3
5 – Código Duplicado	1

c) Considerando a distribuição da Tabela 4.34 e a porcentagem de ocorrências de *refactorings* em relação ao total, a análise de dados foi feita considerando somente as medidas obtidas da classe *Site*.

4.2.5.1. Distribuição da seqüência de *refactorings* por *bad smells*

Tabela 4.36: Distribuição de ocorrências de *refactorings* ao longo do *bad smell* 1 Código Duplicado

Etapa	Refactoring
1	Migração de Trecho de Código para uma Nova Superclasse
2	Migração de Variável para Superclasse
3	Migração de Construtor para Superclasse
4	Migração de Variável para Superclasse
5	Migração de Método para Superclasse
6	Migração de Trecho de Código para um Novo Método
7	Migração de Método para Superclasse
8	Migração de Trecho de Código para um Novo Método
9	Migração de Trecho de Código para um Novo Método
10	Migração de Trecho de Código para um Novo Método
11	Introdução de Método Estrangeiro

Tabela 4.37: Distribuição de ocorrências de *refactorings* ao longo do *bad smell* 2 Código Duplicado / Método Longo

Etapa	Refactoring
12	Migração de Trecho de Código para um Novo Método
13	Migração de Trecho de Código para um Novo Método
14	Migração de Método para Superclasse
15	Dissolução de Variável Temporária
16	Dissolução de Variável Temporária
17	Inclusão de uma Constante para Representar um Número
18	Inclusão de uma Constante para Representar um Número
19	Migração de Trecho de Código para um Novo Método
20	Migração de Trecho de Código para um Novo Método
21	Migração de Variável para Superclasse
22	Migração de Variável para Superclasse
23	Migração de Método para Superclasse
24	Migração de Trecho de Código para um Novo Método (nas classes ResidentialSite e DisabilitySite)
25	Migração de Trecho de Código para um Novo Método Modelo
26	Migração de Trecho de Código para um Novo Método
27	Migração de Trecho de Código para um Novo Método
28	Migração de Método para Superclasse
29	Dissolução de Variável Temporária, Substituição de Parâmetro por Método e Remoção de Parâmetros.
30	Criação da Classe DateRange, do método lastPeriod e aplicação de Substituição de Parâmetro por Método
31	Substituição de Parâmetro por Método

Tabela 4.38: Distribuição de ocorrências de *refactorings* ao longo do *bad smell* 3 Método Longo

Etapa	Refactoring
32	Decomposição de Condicional
33	Criação de dois novos métodos e alteração de um método antigo para chamar os dois novos métodos.
34	Dissolução de Método
35	Migração de Trecho de Código para um Novo Método e Modificação de um método antigo para chamar o novo método.
36	Criar nova classe como extensão de um pacote existente
37	Implementação da interface Cloneable para criação de um clone.
38	Encapsulando Projeção de Tipo
39	Migração de Método para Outra Classe
40	Migração de Método para Outra Classe
41	Criação de quatro métodos
42	Substituição de Algoritmo
43	Migração de Trecho de Código para um Novo Método e Parametrização de Método
44	Migração de Método para Outra Classe
45	Criação de um novo método
46	Criação de um novo método
47	Replace magic number with Constant
48	Criação de dois novos métodos
49	Substituição de Algoritmo
50	Migração de Trecho de Código para um Novo Método
51	Migração de Trecho de Código para um Novo Método
52	Substituição de Algoritmo
53	Criação de um novo método

Tabela 4.39: Distribuição de ocorrências de *refactorings* ao longo do *bad smell* 4 Código Duplicado

Etapa	Refactoring
54	Criar uma subclasse de uma classe já existente
55	Remoção de atributos
56	Remoção de atributos
57	Modificando o método da nova subclasse para chamar um método da superclasse
58	Migração de Método para Superclasse na classe *ResidentialSite* para a classe *Site*; Remover método abstrato da classe *Site*; Remover método da classe *Disability*; Adicionar chamada a um método na classe *LifeLineSite*
59	Migração de Trecho de Código para um Novo Método
60	Migração de Trecho de Código para um Novo Método
61	Substituição de Parâmetro por Método
62	Migração de Trecho de Código para um Novo Método
63	Migração de Trecho de Código para um Novo Método
64	Modificar método usageInRange para usar atributo Integer.MAX_VALUE
65	Mudança de atributo para objeto do tipo *Vector*
66	Criação de novo método

Tabela 4.40: Distribuição de ocorrências de *refactorings* ao longo do Bad Smell 5 Código Duplicado

Etapa	Refactoring
67	Criar uma subclasse de uma classe já existente
68	Modificação em métodos
69	Decompor o método charge para chamar método *fuelCharge* da classe *Site*
70	Migração de Trecho de Código para um Novo Método
71	Migração de Trecho de Código para um Novo Método
72	Alterar método template da superclasse para trabalhar com a nova subclasse
73	Criação de uma nova classe e Migração de Trecho de Código para um Novo Método
74	Migração de Trecho de Código para um Novo Método
75	Migração de Trecho de Código para um Novo Método
76	Migração de Trecho de Código para um Novo Método
77	Renomeando Variável Temporária

4.2.5.2. Identificação das Métricas com maior Variação por Bad Smell

Serão necessários mais estudos de caso para expressar o grau de relacionamento entre *bad smells* e métricas, pois, devido ao número pequeno de ocorrências de *bad smells* reportados no estudo de caso, não foi possível a obtenção de dados representativos para tal finalidade.

4.2.5.3. Identificação dos *Refactorings* que apresentaram Variação por Métrica

Baseando-se no processo de análise representado na Figura 4.1, as tabelas a seguir, resultantes da etapa E, apresentam os *refactorings* que ocasionaram variação para uma dada métrica. Exceto o estudo realizado por Demeyer em (Demeyer e outros, 2000), não conhecemos na literatura nenhum outro estudo que apresente tais informações. Conforme já foi citado na seção 3.4.1, Demeyer propõe heurísticas para a identificação de somente cinco *refactorings* já executados entre diferentes versões de uma dada aplicação (Demeyer e outros, 2000). A abordagem aqui apresentada realizou estudo empírico para fornecer um conjunto de dados que possibilitasse ampliar o escopo do relacionamento entre *refactorings* e métricas, além da possibilidade de relacioná-los com *bad smells*. Além disso, serve também para verificar se o que foi previsto na abordagem *Top Down* ocorreu, avaliar situações não esperadas e identificar quais métricas não apresentaram variação conforme esperado.

Os campos "quantidade de valores fortemente afetados", "quantidade de valores com variação" e "total" são usados para informar o grau de importância da métrica para o *refactoring*. Assim, a métrica será tanto mais sensível ao uso de um determinado *refactoring* quanto maior a relação (quantidade de valores com variação) / (total), onde o campo "total" informa quantas ocorrências do referido *refactoring* envolveram a classe em questão, no nosso caso, a classe *Site*. A esta relação denominamos Coeficiente de Associação entre Métrica e *Refactoring* (CAMR). A relação (quantidade de valores fortemente afetados) / (total) denominamos Coeficiente de Associação Forte entre Métrica e *Refactoring* (CAFMR). Os dois coeficientes podem ser usados para estabelecer o nível de associação entre métricas e *refactorings* variando no intervalo $0 < CAMR < 1$ e $0 < CAFMR < 1$.

Conforme apresentado nas tabelas, a associação entre refactorings e métricas é um artifício a ser usado na redução do volume do código a ser analisado. Este relacionamento pode servir como base para a criação de heurísticas de detecção de

oportunidades de *refactoring* através de uma faixa de valores para um conjunto de métricas pré-determinado.

Tabela 4.41: Identificação das principais variações para a métrica *Attribute Complexity (AC)*

Refactoring afetado pela métrica AC	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Migração de Variável para Superclasse	1	3	4	3/4	¼
Criação de novo método	1	1	2	½	½
Criação de Subclasse	1	1	2	½	½

Tabela 4.42: Identificação das principais variações para a métrica *Cyclomatic Complexity (CC)*

Refactoring afetado pela métrica CC	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Migração de Método para Superclasse	2	2	6	1/3	1/3
Criação de novo método	1	2	2	1	½
Migração de Trecho de Código para uma Nova Subclasse	1	1	2	½	½
Dissolução de Variável Temporária/Substituição de Parâmetro por Método/Remoção de Parâmetros	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método e Parametrização de Método	1	1	1	1*	1*

Tabela 4.43: Identificação das principais variações para a métrica *Halstead Diffficult (HDiff)*

Refactoring afetado pela métrica HDiff	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Migração de Método para Superclasse	2	2	6	1/3	1/3
Substituição de Parâmetro por Método	1	1	1	1*	1*
Dissolução de Método	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método e Parametrização de Método	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método	2	2	2	1	1
Criação de Subclasse	1	1	2	½	½

Tabela 4.44: Identificação das principais variações para a métrica Halstead Effort (HEff)

Refactoring afetado pela métrica Heff	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Migração de Método para Superclasse	1	2	6	1/3	1/6
Dissolução de Variável Temporária/ Substituição de Parâmetro por Método /Remove Paramter	1	1	1	1*	1*
Substituição de Parâmetro por Método	1	1	1	1*	1*
Decomposição de Condicional	1	1	1	1*	1*
Dissolução de Método	1	1	1	1*	1*
Encapsulando Projeção de Tipo	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método e Parametrização de Método	1	1	1	1*	1*
Criação de Subclasse	1	1	2	½	½

Tabela 4.45: Identificação das principais variações para a métrica *Halstead Program Length (HPLen)*

Refactoring afetado pela métrica Hplen	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Migração de Método para Superclasse	2	2	6	1/3	1/3
Migração de Trecho de Código para um Novo Método	1	4	4	1	¼
Dissolução de Variável Temporária/ Substituição de Parâmetro por Método /Remove Paramter	1	1	1	1*	1*
Substituição de Parâmetro por Método	1	1	1	1*	1*
Decomposição de Condicional	1	1	1	1*	1*
Dissolução de Método	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método e Parametrização de Método	1	1	1	1*	1*
Criação de subclasse	1	1	2	½	½
Migração de Trecho de Código para uma Nova Subclasse	1	1	2	½	½

Tabela 4.46: Identificação das principais variações para a métrica *Halstead Program Vocabulary (HPVoc)*

Refactoring afetado pela métrica HPVoc	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Migração de Método para Superclasse	2	2	6	1/3	1/3
Introdução de Método Estrangeiro	1	1	1	1*	1*
Dissolução de Variável Temporária/ Substituição de Parâmetro por Método /Remoção de Parâmetros	1	1	1	1*	1*
Decomposição de Condicional	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método e Parametrização de Método	1	1	1	1*	1*
Criação de novo método	1	2	2	1	½
Migração para uma Nova Subclasse	1	1	2	½	½

Tabela 4.47: Identificação das principais variações para a métrica *Halstead Program Volume (HPVol)*

Refactoring afetado pela métrica HPVol	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Migração de Método para Superclasse	1	2	6	1/3	1/6
Introdução de Método Estrangeiro	1	1	1	1*	1*
Dissolução de Variável Temporária/ Substituição de Parâmetro por Método / Remoção de Parâmetros	1	1	1	1*	1*
Substituição de Parâmetro por Método	1	1	1	1*	1*
Decomposição de Condicional	1	1	1	1*	1*
Dissolução de Método	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método e Parametrização de Método	1	1	1	1*	1*
Migração de Trecho de Código para uma Nova Subclasse	1	1	2	½	½

Tabela 4.48: Identificação das principais variações para a métrica *Lines of Code (LOC)*

Refactoring afetado pela métrica LOC	Quantidade de valores fortemente afetados	Quantidade de Total valores com variação		CAMR	CAFMR
Migração de Método para Superclasse	2	2	6	1/3	1/3
Introdução de Método Estrangeiro	1	1	1	1*	1*
Dissolução de Variável Temporária/ Substituição de Parâmetro por Método /Remoção de Parâmetros	1	1	1	1*	1*
Substituição de Parâmetro por Método	1	1	1	1*	1*
Decomposição de Condicional	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método	2	2	2	1	1
Migração de Trecho de Código para uma Nova Subclasse	1	1	2	½	½

Tabela 4.49: Identificação das principais variações para a métrica *Lack of Cohesion of Methods 1 (LOCOM1)*

Refactoring afetado pela métrica LOCOM1	Quantidade de valores fortemente afetados	Quantidade Total de valores com variação		CAMR	CAFMR
Dissolução de Variável Temporária/ Substituição de Parâmetro por Método /Remoção de Parâmetros	1	1	1	1*	1*
Substituição de Parâmetro por Método	1	1	1	1*	1*
Decomposição de Condicional	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método	2	2	2	1	1

Tabela 4.50: Identificação das principais variações para a métrica *Lack of Cohesion of Methods 2 (LOCOM2)*

Refactoring afetado pela métrica LOCOM2	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Migração de Variável para Superclasse	2	2	4	½	½
Substituição de Parâmetro por Método	1	1	1	1*	1*
Decomposição de Condicional	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método	2	2	2	1	1

Tabela 4.51: Identificação das principais variações para a métrica *Lack of Cohesion of Methods 3 (LOCOM3)*

Refactoring afetado pela métrica LOCOM3	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Migração de Método para Superclasse	1	1	6	1/6	1/6
Migração de Trecho de Código para um Novo Método	3	3	4	¾	¾
Criação de novo método	2	2	2	1	1
Criação de subclasse	1	1	2	½	½

Tabela 4.52: Identificação das principais variações para a métrica *Method Invocation Coupling (MIC)*

Refactoring afetado pela métrica MIC	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Migração de Método para Superclasse	1	1	6	1/6	1/6
Substituição de Parâmetro por Método	1	1	1	1*	1*
Encapsulando Projeção de Tipo	1	1	1	1*	1*
Criação de subclasse	1	1	2	½	½

Tabela 4.53: Identificação das principais variações para a métrica *Maximum Number of Levels (MNOL)*

Refactoring afetado pela métrica MNOL	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Migração de Método para Superclasse	1	1	6	1/6	1/6
Migração de Trecho de Código para um Novo Método e Parametrização de Método	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método	1	1	2	½	½
Criação de Subclasse	1	1	2	½	½

Tabela 4.54: Identificação das principais variações para a métrica *Number of Added Methods (NOAM)*

Refactoring afetado pela métrica NOAM	Quantidade de valores fortemente afetados	Quantidade de Total valores com variação	CAMR	CAFMR
Migração de Método para Superclasse	1	3	6	½	1/6
Dissolução de Variável Temporária, Substituição de Parâmetro por Método e Remoção de Parâmetros	1	1	1	1*	1*
Decomposição de Condicional	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método	2	2	2	1	1

Tabela 4.55: Identificação das principais variações para a métrica *Number of Members (NOM)*

Refactoring afetado pela métrica NOM	Quantidade de valores fortemente afetados	Quantidade de Total valores com variação	CAMR	CAFMR
Migração de Método para Superclasse	1	2	6	1/3	1/6
Dissolução de Variável Temporária, Substituição de Parâmetro por Método e Remoção de Parâmetros	1	1	1	1*	1*
Decomposição de Condicional	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método	2	2	2	1	1

Tabela 4.56: Identificação das principais variações para a métrica *Number of Operations (NOO)*

Refactoring afetado pela métrica NOO	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Migração de Método para Superclasse	1	2	6	1/3	1/6
Dissolução de Variável Temporária, Substituição de Parâmetro por Método e Remoção de Parâmetros	1	1	1	1*	1*
Decomposição de Condicional	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método	2	2	2	1	1

Tabela 4.57: Identificação das principais variações para a métrica Number of Operands (NOprnd)

Refactoring afetado pela métrica Noprnd	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Migração de Método para Superclasse	2	2	6	1/3	1/3
Introdução de Método Estrangeiro	1	1	1	1*	1*
Dissolução de Variável Temporária/Substituição de Parâmetro por Método/ Remoção de Parâmetros	1	1	1	1*	1*
Substituição de Parâmetro por Método	1	1	1	1*	1*
Decomposição de Condicional	1	1	1	1*	1*
Dissolução de Método	1	1	1	1*	1*
Criar nova classe como extensão de um pacote existente	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método e Parametrização de Método	1	1	1	1*	1*
Criação de Subclasse	1	1	2	½	½

Tabela 4.58: Identificação das principais variações para a métrica *Number of Operators (NOprtr)*

Refactoring afetado pela métrica Noprtr	Quantidade de valores fortemente afetados	Quantidade Total de valores com variação	CAMR	CAFMR
Migração de Método para Superclasse	1	2	6	1/3	1/6
Migração de Trecho de Código para um Novo Método	1	1	4	¼	¼
Dissolução de Variável Temporária/Substituição de Parâmetro por Método/ Remoção de Parâmetros	1	1	1	1*	1*
Substituição de Parâmetro por Método	1	1	1	1*	1*
Decomposição de Condicional	1	1	1	1*	1*
Dissolução de Método	1	1	1	1*	1*
Criar nova classe como extensão de um pacote existente	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método e Paramtrização de Método	1	1	1	1*	1*
Criação de Subclasse	1	1	2	½	½

Tabela 4.59: Identificação das principais variações para a métrica *Number of Remote Methods (NORM)*

Refactoring afetado pela métrica NORM	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Migração de Método para Superclasse	1	1	6	1/6	1/6
Encapsulando Projeção de Tipo	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método e Parametrtização de Método	1	1	1	1*	1*
Criação de novo método	1	1	2	½	½

Tabela 4.60: Identificação das principais variações para a métrica *Number of Unique Operands (NUOprnd)*

Refactoring afetado pela métrica NUOprnd	Quantidade de valores fortemente afetados	Quantidade Total de valores com variação	CAMR	CAFMR
Migração de Método para Superclasse	2	2	6	1/3	1/3
Introdução de Método Estrangeiro	1	1	1	1*	1*
Encapsulando Projeção de Tipo	1	1	1	1*	1*
Dissolução de Variável Temporária/Substituição de Parâmetro por Método/Remoção de Parâmetros	1	1	1	1*	1*
Decomposição de Condicional	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método e Paramter Method	1	1	1	1*	1*
Criação de subclasse	1	1	2	½	½

Tabela 4.61: Identificação das principais variações para a métrica *Number of Unique Operators (NUOprtr)*

Refactoring afetado pela métrica NUOprtr	Quantidade de valores fortemente afetados	Quantidade Total de valores com variação	CAMR	CAFMR
Migração de Método para Superclasse	2	2	1/3	1/3
Introdução de Método Estrangeiro	1	1	1*	1*
Criação de classe como extensão de pacote existente	1	1	1*	1*
Migração de Trecho de Código para um Novo Método e Parametrização de Método	1	1	1*	1*
Criação de subclasse	1	1	½	½

Tabela 4.62: Identificação das principais variações para a métrica *Percentage of Private Members (PPrivMr)*

Refactoring afetado pela métrica PprivM	Quantidade de valores fortemente afetados	Quantidade de Total valores com variação	CAMR	CAFMR
Migração de Método para Superclasse	1	1 6	1/6	1/6
Introdução de Método Estrangeiro	1	1 1	1*	1*
Migração de Trecho de Código para um Novo Método	3	4 4	1	3/4
Introdução de Método Estrangeiro	1	1 1	1*	1*
Criação de subclasse	1	1 2	1*	1*

Tabela 4.63: Identificação das principais variações para a métrica *Percentage of Protected Members (PProtM)*

Refactoring afetado pela métrica PprotM	Quantidade de valores fortemente afetados	Quantidade de Total valores com variação	CAMR	CAFMR
Migração de Método para Superclasse	1	2	6	1/3	1/6
Migração de Variável para Superclasse	3	3	4	¾	¾
Migração de Construtor para Superclasse	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método	3	4	4	1	¾
Dissolução de Variável Temporária/ Substituição de Parâmetro por Método/ Remoção de Parâmetros	1	1	1	1*	1*
Criação de subclasse	1	1	2	½	½

Tabela 4.64: Identificação das principais variações para a métrica *Percentage of Public Members (PPubM)*

Refactoring afetado pela métrica PpubM	Quantidade de valores fortemente afetados	Quantidade de Total valores com variação	CAMR	CAFMR
Migração de Método para Superclasse	2	2	6	1/3	1/3
Migração de Variável para Superclasse	1	2	4	½	¼
Migração de Construtor para Superclasse	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método	2	4	4	1	½
Substituição de Parâmetro por Método	1	1	1	1*	1*
Decomposição de Condicional	1	1	1	1*	1*

Tabela 4.65: Identificação das principais variações para a métrica *Response for Class (RFC)*

Refactoring afetado pela métrica RFC	Quantidade de valores fortemente afetados	Quantidade de Total valores com variação	CAMR	CAFMR
Migração de Método para Superclasse	1	2	6	1/3	1/6
Substituição de Parâmetro por Método	1	1	1	1*	1*
Decomposição de Condicional	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método	1	2	2	1	½
Criação de subclasse	1	1	2	½	½

Tabela 4.66: Identificação das principais variações para a métrica *Weighted Methods Per Class 1 (WMPC1)*

Refactoring afetado pela métrica WMPC1	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Migração de Método para Superclasse	2	3	6	½	1/3
Dissolução de Variável Temporária, Substituição de Parâmetro por Método e Remoção de Parâmetros	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método e Parametrização de Método	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método	1	2	2	1	½

Tabela 4.67: Identificação das principais variações para a métrica Weighted Methods Per Class 2 (WMPC2)

Refactoring afetado pela métrica WMPC2	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Migração de Método para Superclasse	1	2	6	1/3	1/6
Dissolução de Variável Temporária, Substituição de Parâmetro por Método e Remoção de Parâmetros	1	1	1	1*	1*
Decomposição de Condicional	1	1	1	1*	1*
Substituição de Parâmetro por Método	1	1	1	1*	1*
Migração de Trecho de Código para um Novo Método	1	2	2	1*	½

4.2.5.4. Identificação das Métricas com maior Variação por Refactoring

A partir das tabelas apresentadas na seção anterior, foram obtidos os seguintes resultados para a identificação das métricas com maior variação por *refactoring*.

Tabela 4.68: Métricas afetadas por Criação de Método

Métrica	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
AC	1	1	2	½	½
CC	1	2	2	1	½
HDiff	2	2	2	1	1
HPVoc	1	2	2	1	½
LOC	2	2	2	1	1
LOCOM1	2	2	2	1	1
LOCOM2	2	2	2	1	1
LOCOM3	2	2	2	1	1
MNOL	1	1	2	½	½
NOAM	2	2	2	1	1
NOM	2	2	2	1	1
NOO	2	2	2	1	1
NORM	1	1	2	½	½
RFC	1	2	2	1	½
WMPC1	1	2	2	1	½
WMPC2	1	2	2	1	½

Tabela 4.69: Métricas afetadas por criação de subclasse

Métrica	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
CC	1	1	2	½	½
HDiff	1	1	2	½	½
Heff	1	1	2	½	½
Hplen	1	1	2	½	½
Hplen	1	1	2	½	½
HPVoc	1	1	2	½	½
HPVol	1	1	2	½	½
LOC	1	1	2	½	½
LOCOM3	1	1	2	½	½
MIC	1	1	2	½	½
MNOL	1	1	2	½	½
Noprnd	1	1	2	½	½
Noprtr	1	1	2	½	½
NUOprnd	1	1	2	½	½
NUOprtr	1	1	2	½	½
PprivM	1	1	2	½	½
PprotM	1	1	2	½	½
RFC	1	1	2	½	½

Tabela 4.70: Métricas afetadas por Decomposição de Condicional

Métrica	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Heff	1	1	1	1*	1*
Hplen	1	1	1	1*	1*
HPVoc	1	1	1	1*	1*
HPVol	1	1	1	1*	1*
LOC	1	1	1	1*	1*
LOCOM1	1	1	1	1*	1*
LOCOM2	1	1	1	1*	1*
NOAM	1	1	1	1*	1*
NOM	1	1	1	1*	1*
NOO	1	1	1	1*	1*
Noprnd	1	1	1	1*	1*
Noprtr	1	1	1	1*	1*
NUOprnd	1	1	1	1*	1*
PpubM	1	1	1	1*	1*
RFC	1	1	1	1*	1*

Tabela 4.71: Métricas afetadas por Encapsulando Projeção de Tipo

Métrica	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Heff	1	1	1	1*	1*
MIC	1	1	1	1*	1*
NORM	1	1	1	1*	1*
NUOprnd	1	1	1	1*	1*

Tabela 4.72: Métricas afetadas por Migração de Trecho de Código para um Novo Método

Métrica	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Hplen	1	4	4	1	¼
LOCOM3	3	3	4	¾	¾
Noprtr	1	1	4	¼	¼
PprivM	3	4	4	1	¾
PprotM	3	4	4	1	¾
PpubM	2	4	4	1	½

Tabela 4.73: Métricas afetadas por Migração de Trecho de Código para um Novo Método e Parametrização de Método

Métrica	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
CC	1	1	1	1*	1*
HDiff	1	1	1	1*	1*
Heff	1	1	1	1*	1*
Hplen	1	1	1	1*	1*
HPVoc	1	1	1	1*	1*
HPVol	1	1	1	1*	1*
NUOprtr	1	1	1	1*	1*
WMPC1	1	1	1	1*	1*
Noprnd	1	1	1	1*	1*
Noprtr	1	1	1	1*	1*
NORM	1	1	1	1*	1*
NUOprnd	1	1	1	1*	1*
MNOL	1	1	1	1*	1*

Tabela 4.74: Métricas afetadas por Dissolução de Método

Métrica	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
HDiff	1	1	1	1*	1*
Heff	1	1	1	1*	1*
Hplen	1	1	1	1*	1*
HPVol	1	1	1	1*	1*
Noprnd	1	1	1	1*	1*
Noprtr	1	1	1	1*	1*

Tabela 4.75: Métricas afetadas por Dissolução de Variável Temporária/Substituição de Parâmetro por Método/Remoção de Parâmetros

Métrica	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
NOAM	1	1	1	1*	1*
NOM	1	1	1	1*	1*
NOO	1	1	1	1*	1*
WMPC1	1	1	1	1*	1*
WMPC2	1	1	1	1*	1*
Heff	1	1	1	1*	1*
Hplen	1	1	1	1*	1*
HPVoc	1	1	1	1*	1*
HPVol	1	1	1	1*	1*
LOC	1	1	1	1*	1*
LOCOM1	1	1	1	1*	1*
PprotM	1	1	1	1*	1*
Noprnd	1	1	1	1*	1*
Noprtr	1	1	1	1*	1*
CC	1	1	1	1*	1*
NUOprnd	1	1	1	1*	1*

Tabela 4.76: Métricas afetadas por Introdução de Método Estrangeiro

Métrica	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Noprnd	1	1	1	1*	1*
NUOprnd	1	1	1	1*	1*
NUOprtr	1	1	1	1*	1*
PprivM	1	1	1	1*	1*
PprivM	1	1	1	1*	1*
HPVoc	1	1	1	1*	1*
HPVol	1	1	1	1*	1*
LOC	1	1	1	1*	1*

Tabela 4.77: Métricas afetadas por Migração de Construtor para Superclasse

Métrica	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
PprotM	1	1	1	1*	1*
PpubM	1	1	1	1*	1*

Tabela 4.78: Métricas afetadas por Migração de Variável para Superclasse

Métrica	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
AC	1	3	4	¾	¼
LOCOM2	2	2	4	½	½
PprotM	3	3	4	¾	¾
PpubM	1	2	4	½	¼

Tabela 4.79: Métricas afetadas por Migração de Método para Superclasse

Métrica	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
CC	2	2	6	1/3	1/3
HDiff	2	2	6	1/3	1/3
Heff	1	2	6	1/3	1/6
Hplen	2	2	6	1/3	1/3
HPVoc	2	2	6	1/3	1/3
HPVol	1	2	6	1/3	1/3
LOC	2	2	6	1/3	1/3
LOCOM3	1	1	6	1/6	1/6
MIC	1	1	6	1/6	1/6
MNOL	1	1	6	1/6	1/6
NOAM	1	3	6	½	1/6
NOM	1	2	6	1/3	1/6
NOO	1	2	6	1/3	1/6
Noprnd	2	2	6	1/3	1/3
Noprtr	1	2	6	1/3	1/6
NORM	1	1	6	1/6	1/6
NUOprnd	2	2	6	1/3	1/3
NUOprtr	2	2	6	1/3	1/3
PprivM	1	1	6	1/6	1/6
PprotM	1	2	6	1/3	1/6
PpubM	2	2	6	1/3	1/3
RFC	1	2	6	1/3	1/6
WMPC1	2	3	6	½	1/3
WMPC2	1	2	6	1/3	1/6

Tabela 4.80: Métricas afetadas por Substituição de Parâmetro por Método

Métrica	Quantidade de valores fortemente afetados	Quantidade de valores com variação	Total	CAMR	CAFMR
Hdiff	1	1	1	1*	1*
Heff	1	1	1	1*	1*
Hplen	1	1	1	1*	1*
HPVol	1	1	1	1*	1*
LOC	1	1	1	1*	1*
LOCOM1	1	1	1	1*	1*
LOCOM2	1	1	1	1*	1*
MIC	1	1	1	1*	1*
Noprnd	1	1	1	1*	1*
Noprtr	1	1	1	1*	1*
PpubM	1	1	1	1*	1*
RFC	1	1	1	1*	1*
WMPC2	1	1	1	1*	1*

4.2.6. Análise dos Resultados

O uso do processo de análise de dados relatado na Figura 4.1 teve como conjunto de resultados do estudo de caso da abordagem *Bottom Up* as seções 4.5.3 (*refactorings* que apresentaram variação por métrica) e 4.5.4 (métricas com maior variação por refactoring). Dos 13 *refactorings* que ocasionaram variação das métricas, 8 apresentaram somente uma ocorrência, fato que os tornam, neste estudo de caso, pouco representativos para fins de análise. Será apresentada a seguir a relação dos *refactorings* cuja aplicação resultou em variação das métricas com somente uma ocorrência: Substituição de Parâmetro por Método, Migração de Construtor para Superclasse, Introdução de Método Estrangeiro, Dissolução de Variável Temporária/Substituição de Parâmetro por Método/Remoção de Parâmetros, Dissolução de Método, Migração de Trecho de Código para um Novo Método e Parametrização de Método, Encapsulando Projeção de Tipo, Decomposição de Condicional. Em relação aos outros 5 *refactorings* (Migração de Método para Superclasse, Migração de Variável para Superclasse, Migração de Trecho de Código para um Novo Método, Criação de Subclasse, Criação de Método), verificou-se que o número de métricas associadas ao *refactoring* ocorre conforme distribuição da Tabela 4.81.

Tabela 4.81: Relacionamento entre *refactorings*, métricas e número de ocorrências de *refactorings*

Refactoring	Número de métricas associadas	Número de ocorrências do refactoring
Migração de Método para Superclasse	24	6
Migração de Variável para Superclasse	4	4
Migração de Trecho de Código para um Novo Método	6	4
Criação de Subclasse	18	2
Criação de Método	16	2

Pela distribuição apresentada, verifica-se que os resultados com maior número de ocorrências de *refactoring* trarão maior confiabilidade no que tange aos Coeficientes de Associação entre Métricas e *Refactorings* (CAMR e CAFMR). Reconhecemos a necessidade da realização de um número maior de estudos de caso para que seja alcançado um número apropriado de ocorrências dos *refactorings* e, conseqüentemente, maior representatividade nos resultados obtidos.

5. CONSIDERAÇÕES FINAIS

Esta dissertação apresenta uma metodologia que introduz métricas na aplicação de *refactorings*. Ela argumenta que, partindo da análise de código fonte, há a possibilidade de redução da dependência da análise cognitiva através do estabelecimento de conjuntos de métricas relacionados com tipos específicos de *refactorings*. Como um passo inicial no sentido do estabelecimento desta metodologia, esta dissertação estuda duas abordagens para relacionar métricas, *refactorings* e *bad smells*:

a) Abordagem *Top Down*: usa o método Meta Pergunta Métrica segundo análise teórica.

b) Abordagem *Bottom Up*: método elaborado nesta dissertação para execução de análise empírica em uma aplicação ao longo da seqüência de 77 *refactorings*.

O resultado das duas abordagens, apesar das limitações verificadas nos estudos de caso, comprova que métricas podem auxiliar no uso de *refactorings*. A abordagem *Top Down* constatou, conforme apresentado na Tabela 4.33 da Seção 4.1.3, que as métricas disponíveis representam 25% do conjunto de métricas (somatório dos percentuais das métricas dos tipos 1 e 2), enquanto que as métricas não disponíveis representam os restantes 75% do conjunto de métricas. Este último valor foi obtido da soma dos seguintes percentuais:

- 58,33% são factíveis e implementáveis (somatório dos percentuais das métricas dos tipos 3, 4 6).;
- 16,67% são métricas não disponíveis ou não difundidas (métricas do tipo 5), cujo resultado depende fortemente de análise cognitiva e de difícil implementação.

A abordagem *Bottom Up* constatou, através do processo de análise dos dados apresentado na Figura 4.1 da Seção 4.2.4.1, a existência de relacionamento entre métricas e *refactorings* relatadas nas diversas tabelas presentes na Seção 4.2.5.4. Ainda na abordagem *Bottom Up*, foi informado, na Seção 4.2.5.2, que serão necessários mais estudos de caso para expressar o grau de relacionamento entre *bad smells* e métricas, pois, devido ao número pequeno de ocorrências de *bad smells* reportados no estudo de caso, não foi possível a obtenção de dados representativos para tal finalidade.

Da mesma forma argumentada por Fowler (2000a), a metodologia proposta nesta dissertação indica que a última e definitiva etapa para a decisão sobre a aplicação de um determinado *refactoring* é a análise cognitiva. Entretanto, para se chegar a esta última etapa, métricas, conforme indicado pelas abordagens *Top Down* (somente 16,67% das métricas obtidas como importantes para o diagnóstico de oportunidades de refactoring são fortemente dependentes de análise cognitiva e de difícil implementação) e *Bottom Up* (através da existência de relacionamento entre métricas e *refactorings* relatados pelos valores representativos dos Coeficientes de Associação entre Métrica e *Refactoring* – CAMR – e Coeficientes de Associação Forte entre Métrica e *Refactoring* – CAFMR - nas diversas tabelas presentes na seção 4.2.5.4), representam um aspecto facilitador fundamental nas etapas do uso de *refactoring*.

Verificou-se também, ao longo do estudo de caso, que para um determinado *bad smells* podem ser aplicadas mais de uma seqüência de refactorings, fato que pode resultar em valores medidos diferentes ao longo do processo a depender da alternativa adotada.

5.1. CONTRIBUIÇÕES ESPERADAS DA NOSSA METODOLOGIA

Uma vez implementada, a metodologia proposta na Seção 3.4 dará as seguintes contribuições:

a) Redução do volume de código fonte a ser analisado pela equipe de *refactoring*.
b) Criação de base concreta para analisar a conveniência ou não de tratar um *bad smell*.
c) Possibilidade de relacionamento entre a análise cognitiva tradicionalmente feita antes do *refactoring* e a avaliação empírica possibilitada pelas métricas utilizadas para quantificar *refactorings*.

5.2. CONTRIBUIÇÕES DA DISSERTAÇÃO

A principal contribuição desta dissertação é propor duas abordagens para relacionar métricas, *refactorings* e *bad smells* e realizar os estudos de caso para avaliar estas abordagens. Além disso, podem ser citadas as seguintes contribuições:

a) Uso da abordagem Meta Pergunta Métrica (MPM) para indicar métricas completamente novas com o objetivo de quantificar *bad smells* e *refactorings*.
b) Indicação de métricas disponíveis para quantificar *bad smells* e *refactorings.*

c) Possibilidade de pesquisadores da área de medição de software utilizarem as heurísticas desenvolvidas por pesquisadores da área de refactoring na identificação de novas métricas para avaliação do código.

5.3. LIMITAÇÕES

Este trabalho propôs uma metodologia e deu um passo inicial na direção de sua implementação. Para consolidar tal implementação serão necessários, dentre outras, as seguintes atividades:

a) Realização de mais estudos de caso para que seja criada uma base de conhecimento mais ampla. Esta base de conhecimento será útil para consolidar os relacionamentos entre *refactorings*, *bad smells* e métricas.
b) A implementação das métricas identificadas pelo paradigma Meta Pergunta Métrica (MPM).

O foco desta dissertação é a análise da metodologia proposta aplicada a programas escritos em Java. Os resultados obtidos não foram testados ou comparados com aplicações escritas em outras linguagens orientadas a objeto.

A avaliação das abordagens propostas ficou limitada aos *refactorings* e *bad smells* detectados no estudo de caso, não sendo possível, portanto, abrangência a todos aqueles propostos por Fowler.

Em relação à metodologia apresentada no capítulo 3, não foi possível estabelecer um conjunto de métricas a ser usado para a identificação dos *bad smells* pelo fato de grande parte das mesmas ainda não estarem difundidas na literatura.

5.4. TRABALHOS FUTUROS

a) Elaboração de metodologia para auxílio na decisão de quando iniciar e parar o refactoring. Segundo Fowler (2000a), estes aspectos são tão importantes quanto saber aplicá-lo e como operar os seus mecanismos.
b) Derivação das métricas selecionadas na abordagem *Top Down* e uso das mesmas empiricamente para *refactoring*.
c) Estabelecimentos de novos critérios que possibilitem reduzir a necessidade de análise cognitiva no processo de detecção de *bad smells* e na escolha das possíveis seqüências de *refactorings* a serem adotadas.

d) Estabelecimento de critérios para a definição de faixas de valores aceitáveis para as métricas indicadas nas abordagens *Top Down* e *Bottom Up* através de estudos de caso.

6. REFERÊNCIAS BIBLIOGRÁFICAS

Astels, D. **Refactoring with UML**. XP 2002. Disponível em <www.xp2002.org_atti_DaveAstels--RefactoringWithUML.pdf>. Acesso em 02 de agosto de 2002.

Basili, V., Rombach H. **The TAME project: Towards improvement-oriented software environments**. IEEE Transactions on Software Eng., June, 1988.

Basili, V., Weiss, D. **A methodology for collecting valid software engineering data**. IEEE Transactions on Software Eng., November 1984.

Beck, K. **Extreme Programming Explained: Embrace Change**. Addison Wesley, 1999.

Beck, K., Gamma, E. **JUnit Open-Source Testing Framework**. Disponível em <www.junit.org>. Acesso em 25 de março de 2002.

Briand, L. **Measurement and Quality Modelling of OO Systems**. Software Metrics Symposium. The Sixth International Symposium on Software Metrics. November 4-6, 1999. Boca Raton, Florida, USA

Carlos, S. C. **The Elimination of Overheads due to Type Annotations and the Identification of Candidate Refactorings**. Master Thesis. North Carolina State University, 2002.

Carneiro, G. F., Mendonça Neto, M.G. **Usando Medição de Código Fonte para Refactoring**. In 2nd Ibero-American Symposium on Software Engineering and Knowledge Engineering. Oct, 2002. Salvador, Bahia, Brazil.

Chidamber, S., Kemerer, C. **Metrics Suite for Object Oriented Dedsign**. IEEE Transactions on Software Engineering, vol.20, n.6, June 1994.

Cinnéide, M., Nixon, P. **Composite Refactorings for Java Programs**. In ECOOP 2000.

CodeMorpher. Disponível em <http://www.xptools.com/?doc=products>. Acesso em 30 de abril de 2002.

Demeyer, S., Ducasse, S., Nierstrasz, O. **Finding Refactorings via Change Metrics**. Proceedings OOPSLA´2000, ACM Press.

Dudziak, T., Wloka, J. **Tool-Supported Discovery and Refactoring of Structural Weaknesses in Code**. Diploma Thesis of the Faculty of Computer Science Technical University of Berlin. February, 2002.

Eclipse. Disponível em <http://www.eso.org/~hsommer/eclipse20020923.htm>. Acesso em 01 de outubro de 2002.

Emden, E. e Moonen, L. **Java Quality Assurance by Detecting Code Smells**. WCRE 2002 - Working Conference on Reverse Engineering October 29 - November 1, 2002. Richmond, Virginia, USA.

Ernst, M. D. **Dynamically Discovering Likely Program Invariants**. PhD Thesis, University of Washington. Department of Computer Science and Engineering, Seatle, Washington, August 2000.

Ernst, M. D, Czeisler, A., Griswold, W. G., Notkin, D. **Quickly detecting relevant program invariants**. In International Conference on Software Engineering, pages 449-458, 2000.

Ernst, M. D., Cockrell J., Griswold, W. G., David Notkin. **Dynamically discovering likely program invariants to support program evolution**. IEEE Transactions on Software Engineering, 27(2):1-25, February 2001.

Fabry, J., Mens, T.. **Language-independent Detection of OO Paterns using Logic Meta-Programming**. Fifth International Symposium on Practical Aspects of Declarative Languages (PADL 2003) New Orleans, Louisiana, USA 13-14. Jan 2003.

Fenton, N. and Pleeger S. **Software Metrics: A Rigorous and Pratical Approach**. Second Edition, PWS Publishing Company, 1997.

Fowler, M. **Refactoring: Improving the Design of Existing Code**. Addison-Wesley, 2000a.

Fowler, M. **Capítulo 15 não publicado no livro Refactoring: Improving the Design of Existing Code**. Addison-Wesley, 2000b, mas disponibilizado pelo autor em http://www.refactoring.com/rejectedExample.pdf. Documento acessado em 20/03/2002.

Gamma, E., Helm R., Johnson R., Vlissedes J.. **Design Patterns: Elements of Object-Oriented Software**. Addison Wesley, 1995.

Griffiths, M. **Retool for JBuilder**. Disponível em <http://www.elists.org/pipermail/jbuilder/2001-November/000139.html>. Acesso em 21 de novembro de 2001.

Griswold, W and Notkin, D.. **Automated assistance for program restructuring**. ACM Transactions on Software Engineering and Methodology, 2(3):228-269, July 1993.

Griswold, W. **Program Restructuring as an Aid to Software Maintenance**. PhD Thesis, University of Washington, Dept. of Computer Science & Enginnering, August 1991. Technical Report No. 91-08-04.

Hudli, R. Hoskings, C., Hudli, A. **Software Metrics for Object Oriented Designs**, IEEE 1994.

Intellij. Disponível em <http://www.intellij.com/idea/features/features.jsp>. Acesso em 15 de dezembro de 2002.

JavaRefactor. Disponível em <http://plugins.jedit.org/plugins/?JavaRefactor>. Acesso em 20 de julho de 2002.

JBuilder. Disponível em <http://www.borland.com/jbuilder/>. Acesso em 02 de janeiro de 2003.

Kataoka, Y., Ernst, M., Griswold, W., Notkin, D.. **Automated Support for Program Refactoring using Invariants**. In ICSM'01, Proceedings of the International Conference on Software Maintenance, (Florence, Italy), November 6-10, 2001, pp. 736-743.

Lee, Y., Liang, B., Wang, F. **Some Complexity Metrics for Object Oriented Programs Based on Information Flow**. Proceedings: CompEuro, March 1993, pp: 302-310

Lewerentz, C., Rust, H., Simon F. **Quality – Metrics – Numbers – Consequences: Lessons Learned**. In Reiner Dumke, Franz Lehner (Hrsg): "Software – Metriken: Entwicklungen, Werkzeug und Anwendungsverfahren", p. 51-70, Gabler Verlag, 2000.

Lorenz, M. and Kidd, J. **Object-Oriented Software Metrics: A Pratical Approach**. Prentice-Hall, 1994.

Marinescu, R. **An Object Oriented Metrics Suite on Coupling**. University Politehnica Timisoara, Facultatea de Automatica si Calculatoare, Departamentul de Calculatoare si Inginerie Software. September, 1998.

McCabe & Associates, McCabe **Object Oriented Tool User´s Instructions**, 1994.

McCabe, T. **A software complexity measure**. IEEE Transactions on Software Engineering SE-2(4), 1976.

Mendonça, M.. **An Approach to Improving Existing Measurement Frameworks in Software Development Organizations**. Ph.D. Thesis, University of Maryland College Park, 1997.

Mendonça, M., Basili, V., Bhandari, I., Dawson, J.. **An Approach for Improving Existing Measurement Frameworks**. IBM Systems Journal, v.37, n.4, p.484-501, 1998.

Mendonça, M., Basili, V., **Validation of an Approach for Improving Existing Measurement Frameworks**. IEEE Transactions on Software Engineering, v.26, n.6, p.484-499, 2000.

Norda, J. **A Refactoring Tool for Java**. Master Thesis. Linköpings Universitet. 26-Fev-2001.

Opdyke, W. **Refactoring Object-Oriented Frameworks**. Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1992.

Opdyke, W., Johnson, R.. **Refactoring: An aid in designing application frameworks and envolving object-oriented systems**. I Proceedingsof SOOPPA ´90 Symposium on Object-Oriented Programming Emphasizing Practical Applications, Sep 1990.

RefactorIt. Disponível em <http://www.refactorit.com>. Acesso em 10 de dezembro de 2002.

Roberts, D. **Practical Analysis for Refactoring**. PhD Thesis.University of Illinois at Urbana-Champaign, 1999.

Simon, F., Steinbrickner, F., C. Lewerentz. **Metrics based refactoring**, in: Proc. European Conf. Software Maintenance and Reengineering (2001), pp. 30- 38.

Sölingen, R. and Berghout, E. **The Goal/Question/Metric Method: a pratical guide for improvement of software development**. McGraw-Hill International (UK), 1999.

Sÿsta, T. **Static and Dynamic Reverse Engineering Techniques for Java Software Systems**. PhD Thesis. University of Tampere. Department of Computer and Information Science. Finland. May, 2000

Thompson, S., Reinke, C.. **Refactoring Functional Programs**. Technical Report 16-01, Computing Laboratory, University of Kent at Canterbury, October 2001.

TogetherControlCenter. Acesso em 03 de julho de 2002. Disponível em <http://www.togethersoft.com/products/controlcenter/index.jsp>.

Tourwé, T., Brichau, J., Mens, T. **Using Declarative Metaprogramming to Detect Possible Refactorings**. ASE 2002. Disponível em <http://www.cs.ubc.ca_~kdvolder_Workshops_ASE2002_DMP_papers_08tourwe-brichau-mens.pdf>. Acesso em 6 de setembro de 2002.

Transmogrify. Disponível em <http://transmogrify.sourceforge.net>. Acesso em 12 de dezembro de 2001.

Wuyts, R.. **A logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation**. PhD Thesis, Department Informatica, Vrije Universiteit Brussel, 2001.

Xrefactory. Disponível em <http://transmogrify.sourceforge.net/>. Acesso em 02 outubro 2002.

APÊNDICE 01 – MÉTRICAS DISPONÍVEIS NO *TOGETHER CONTROL CENTER 6*

O *Together Control Center* versão 6.0 (2002) disponibiliza as seguintes métricas utilizadas no estudo de caso.

1. AC - *Attribute Complexity*

Definido como a soma do valor de cada atributo na classe. Usa-se "*" para definir tipos de pacote incluindo todos os seus subpacotes. Por exemplo, *java.lang.** significa que todas as classes do pacote *java.lang* e seus subpacotes serão contemplados. Para processar todos os tipos listados na tabela, especifique a última linha como "*". A ordem da linha é importante, pois a verificação dos atributos é realizada de cima para baixo. Repetições de tipo não são contabilizadas, assim se um tipo mais específico está contido em um outro mais genérico já verificado, o mesmo não voltará a ser contabilizado. Exemplo, *java.lang.** não será contabilizado, se vier após *java.**

2. CBO - *Coupling Between Objects*

Representa o número de outras classes com as quais a classe em questão está acoplada. Contabiliza o número de referencias a tipos que é utilizado na declaração de atributos, parâmetros formais, declarações com lançamento de exceção e variáveis locais a partir dos quais a seleção dos métodos e atributos é realizada. Tipos primitivos, tipos do pacote *java.lang* e supertipos não são contabilizados.

Acoplamento excessivo entre objetos é prejudicial à modularização do projeto e dificulta o reuso. Quanto mais independente for a classe, mais fácil será a sua reutilização. Para que seja proporcionada a modularidade e o encapsulamento, o aclopamento entre objetos de classes diferentes deve ser o mínimo possível. Quanto maior o número de acoplamentos, maior a sensibilidade a mudanças em outras partes do projeto, fato que torna a manutençao mais difícil. Uma medida de acoplamento é útil para determinar quão complexo será o teste das diversas partes do projeto. Quanto maior o acoplamento entre objetos de classes diferentes, mais rigoroso deverá ser o teste.

3. CC - *Cyclomatic Complexity*

Esta medida representa a complexidade de uma classe. Contabiliza o número de caminhos possíveis em um algoritmo.

A definição de CC introduzida por McCabe (1976) considera a forma do diagrama de fluxos de um programa como a medida da sua complexidade: CC = L - N + 2P onde L é o número de *links* do grafo, N é o número de nós e P é o número de partes desconectadas do grafo. Por exemplo, considere um método que consiste da seguinte sentença if:

```
if (x>0) {
   x++;
} else {
   x--;
}
```

Aplicando a fórmula CC = L - N + 2P (A1.1)

CC = 4 - 4 + 2*1 = 2

Uma outra forma menos formal de calcular é CC = D + 1 (A1.2) onde D é o número de decisões binárias do grafo, se ele tiver uma entrada e uma saída. Em outras palavras, o número de if, for e while, além do número de operações lógicas “and” e “or”. Para o exemplo acima: CC = D + 1 = 1 + 1 = 2.

Cyclomatic complexity (McCabe & Associates, McCabe Object Oriented Tool User´s Instructions, 1994) é usada para avaliar a complexidade de um algoritmo usado em um método. É contabilizado o número de casos de testes necessários para testar o método. A fórmula para o cálculo de *cyclomatic complexity* é o número de *links* do grafo menos o número de nodos somado de duas unidades. Para uma sequência onde há somente um caminho de fluxo, sem escolha ou opção, apenas um caso de teste é nencessário. Um if, entretanto, tem duas opções, se a condiçção for verdadeira, um caminho será selecionado, se a condição for falsa, um caminho alternativo será testado. A figura A.1 mostra exemplos de cálculo de *cyclomatic complexity* para quatro

estruturas de programação básicas.(McCabe & Associates, McCabe Object Oriented Tool User's Instructions, 1994).

Deseja-se um método com um baixo cyclomatic complexity. Cyclomatic complexity não pode ser usada para medir a complexidade de uma classe por causa de herança, mas cyclomatic complexity de métodos individuais pode ser combinada com outras medidas para que seja avaliada a complexidade de uma classe. Apesar desta métrica estar especificamente associada à avaliação de complexidade, ela também está relacionada a outros atributos (Hudli e outros, 1994) (Lee e outros, 1993) (Lorenz e outros, 1994)) (McCabe e outros, 1994), (Tegarden e outros, 1992).

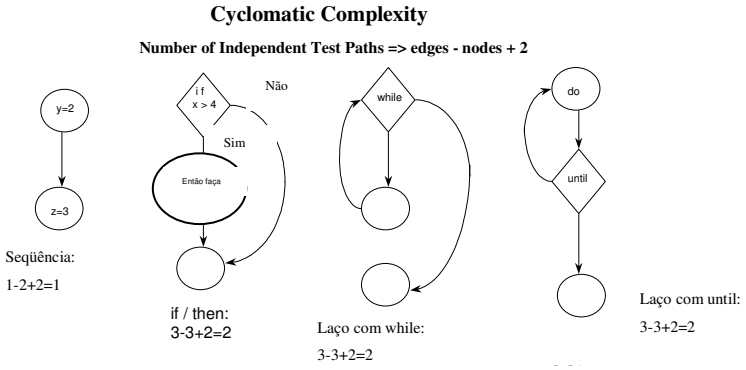


Figura A.1: Exemplos de cálculo da métrica *Cyclomatic Complexity*

4. CR - *Comment Ratio*

Contabiliza a proporção entre o número de comentários e o número total de linhas de código (sendo os comentários incluídos no número de linhas do código).

5. DOIH – *Depth Of Inheritance Hierarchy*

Contabiliza quão profunda a hierarquia de herança de uma classe ou interface é declarada. Valores altos significam que a classe tem alto nível de especialização.

6. Hdiff – *Halstead Difficulty*

É uma medida do tipo *Halstead Software Science metrics*. É calculada como Hdiff = ('Número de Operadores Únicos' / 2) * ('Número de operandos' / 'Número de operandos únicos') (A1.3).

7. HEff - *Halstead Effort*

É uma medida do tipo *Halstead Software Science metrics*. É calculada como HEff = (*Halstead Difficulty)* * (*Halstead Program Volume)*. (A1.4)

8. HPLen - *Halstead Program Length*

É uma medida do tipo *Halstead Software Science metrics*. É calculada como HPLen = (Número de operadores) + (Número de Operandos). (A1.5)

9. HPVoc - *Halstead Program Vocabulary*

É uma medida do tipo *Halstead Software Science metrics*. É calculada como HPVoc = (Número de operadores únicos) + (Número de Operandos únicos). (A1.6)

10. HPVol - *Halstead Program Volume*

É uma medida do tipo *Halstead Software Science metrics*. É calculada como HPVol = (*Halstead Program Length)* * Log2(*Halstead Program Vocabulary*) (A1.7).

11. LOC - *Lines Of Code*

Esta é uma medida tradicional do tamanho do código. Conta o número de linhas.

12. LOCOM1 - *Lack Of Cohesion Of Methods 1*

Analisa cada par de métodos em uma classe e determina o conjunto de campos que cada uma acessa. Se existem conjuntos disjuntos de acessos aos campos, o contador P aumenta de uma unidade. Se eles têm pelo menos um campo de acesso, Q aumenta de uma unidade. Após considerar cada par de métodos tem-se:

RESULT = (P > Q) ? (P - Q) : 0 (A1.8)

Um valor baixo indica alto acoplamento entre os métodos, resultando em grande esforço de teste tendo em vista que muitos métodos podem afetar os mesmos atributos. Isto indica baixo índice de reuso. (Chidamber e outros, 1994).

13. LOCOM2 - *Lack Of Cohesion Of Methods 2*

Conta a porcentagem de métodos que não acessam um atributo específico em relação a todos os atributos da classe. Um alto valor de coesão implica em uma classe bem projetada. Uma classe coesiva proporcionará um alto grau de encapsulamento, sendo que a perda de coesão diminuirá o grau de encapsulamento e aumentará a complexidade.

14. LOCOM3 - *Lack Of Cohesion Of Methods 3*

Mede a dissimilaridade dos métodos em uma classe pelos atributos. Considere um conjunto de métodos ***m***, M_1 , M_2, ... , M_m

Os métodos acessam os atributos de dados ***a***, A_1, A_2, ... , A_a

Fazendo-se $a(M_k)$ = número de atributos acessados pelo método M_k

Fazendo-se $m(A_k)$ = número de métodos que acessam o dado A_k

Então,

$$LOCOM3 = \frac{1 / a \sum_{i=1}^{a} m(A_i) - m}{1 - m} 100 \qquad \text{(A1.9)}$$

Valores baixos indicam boa subdivisão da classe, implicando em simplicidade e alto reuso. Valores altos correspondem em aumento da complexidade, consequentemente aumentando a probabilidade de erros durante o processo de desenvolvimento.

15. MIC - *Method Invocation Coupling*

Obtém o número relativo de outras classes para as quais uma determinada classe envia mensagens.

Definição. $MIC_{norm} = n_{MIC} / (N - 1)$ (A1.10)

onde N é o número de classes definidas no projeto e n_{MIC} é o número de classes para as quais mensagens são enviadas(Marinescu, 1998).

Pontos de vista: Os pontos de vista a seguir resumem o impacto que o acoplamento tem em alguns atributos externos.

- **Manutenibilidade.** Uma classe fortemente acoplada (com valor alto da métrica MIC) é mais difícil de manter devido a dependência em relação às classes com as quais está acoplada(Marinescu, 1998).
- **Compreensibilidade.** Uma classe com forte acoplamento é de difícil compreensão, tendo em vista que implica em compreensão parcial e, às vezes, total das classes com as quais está acoplada(Marinescu, 1998).
- **Testabilidade e vulnerabilidade a erros.** Uma classe está sujeita a erros na mesma proporção que o número de classes às quais está acoplada.

Assim, alto fator de acoplamento influencia negativamente a testabilidade (Marinescu, 1998).

16. MNOL - *Maximum Number Of Levels*

Conta a máxima profundidade do `if`, `for` and `while` ao longo dos métodos. Unidades lógicas com um alto número de níveis encadeados precisam de simplificação de implementação e melhoria do processo, isto porque grupos que contêm mais de sete partes de informação são consideravelmente mais difíceis de compreender quando da resolução de problemas.

17. MNOP - *Maximum Number Of Parameters*

Informa, através da comparação do número de parâmetros de todas as operações em uma classe, o maior número de parâmetros possíveis. Métodos com muitas operações tendem a ser mais especializados, diminuindo as possibilidades de reuso.

18. NOAM - *Number Of Added Methods*

Informa o número de operações novas adicionadas por uma classe. Operações herdadas e sobrescritas não são consideradas. Um alto valor desta métrica indica que a funcionalidade de uma dada classe está se tornando distinta das suas classes ancestrais.

19. NOM - *Number Of Members*

Informa o número de membros (atributos e operações/métodos) de uma classe. Membros herdados podem ser opcionalmente considerados no total. Se uma classe tem um alto número de membros, pode-se avaliar a possibilidade de dividí-la em duas ou mais subclasses.

20. NOO - *Number Of Operations*

Informa o número de operações. Membros herdados podem ser considerados. Se uma classe tem um alto número de operações, pode-se avaliar a possibilidade de dividi-la em duas ou mais subclasses.

21. NOprnd - *Number of Operands*

Esta métrica é usada como entrada para as métricas do tipo *Halstead Software Science*. Contabiliza o número de operandos usados numa classe.

Exemplo:

```
01 public class Class1 {
02     public void x(boolean v) {
03         int i;
04         if (v) {i = 1;}
05         else {i = 2;}
06         switch(i){
07             case 1:
08             case 2:
09             default:;
10         }
11         try {
12             while(v){
13                 v = false;
14                 int r = 1;
15                 Boolean b = new Boolean(true);
16                 i = i*i+r;
17                 break;
18             }
19         } catch (Exception e) {
20             throw e;
21         }
22     }
23 }
```

Linha	N1	n1	N2	n2
04	if, =	if, =	v, i, 1	v, I, 1
05	=		i, 2	2
06	switch	switch	i	
12	laço	laço	v	
13	=		v, false	false
14	=		1	
15	=, new, call	new, call	Boolean, true	Boolean, true
16	=, *, +	*, +	i, i, i, r	r
20	throw	throw	e	e
Total	**14**	**9**	**17**	**9**

Onde,

N1 é NOprtr (Número de operadores),

n1 é NUOprtr (Número de operadores únicos),

N2 é NOprnd (Número de operandos),

n2 é NUOprnd (Número de operandos únicos)

22. NOprtr - *Number of Operators*

Esta métrica é usada como entrada para as métricas do tipo Halstead Software Science. Contabiliza o número de operadores usados numa classe.

23. NORM - *Number of Remote Methods*

Contabiliza o número de métodos remotos chamados. Como método remoto considera-se aquele que não é declarado na classe em questão nem em seus ancestrais.

24. NUOprnd - *Number of Unique Operands*

Usada como entrada para as métricas do tipo Halstead Software Science. Contabiliza o número de operandos únicos usados na classe.

25. NUOprtr - *Number of Unique Operators*

Usada como entrada para as métricas do tipo Halstead Software Science. Contabiliza o número de operadores únicos usados na classe.

26. PPrivM - *Percentage of Private Members*

Contabiliza a porcentagem de membros privados na classe.

27. PProtM - *Percentage of Protected Members*

Contabiliza a porcentagem de membros protegidos na classe.

28. PPubM - *Percentage of Public Members*

Contabiliza a proporção de membros vulneráveis em uma classe. Um grande proporção destes membros siginifica que a classe tem maior potencial de ser afetado por classes externas, significando que maior esforço será necessário para testar a classe em questão.

29. RFC - *Response for Class*

O tamanho do conjunto de resposta para uma classe inclui também os métodos herdados da hierarquia da classe e os métodos que podem ser chamados por outros objetos. Uma classe que proporciona um conjunto de resposta grande é considerada mais complexa e requer mais esforços nos testes que outra de menor complexidade. Esta medida é calculada como a soma do número de métodos locais e métodos remotos.

30. TCR - *True Comment Ratio*

Contabiliza a proporção de documentação e ou comentários de implementação com o total de linhas de código (todos os comentários são excluídos deste valor).

31. WMPC1 - *Weighted Methods Per Class 1*

Esta métrica é a soma da complexidade de todos os métodos da classe, onde o peso de cada método é associado a sua complexidade (cyclomatic complexity). O número de métodos e suas respectivas complexidades podem ser úteis para o prognóstico de quanto tempo e esforço será necessário para desenvolver e manter a classe. Apenas os métodos especificados na classe são incluídos, assim qualquer método herdado não será considerado.

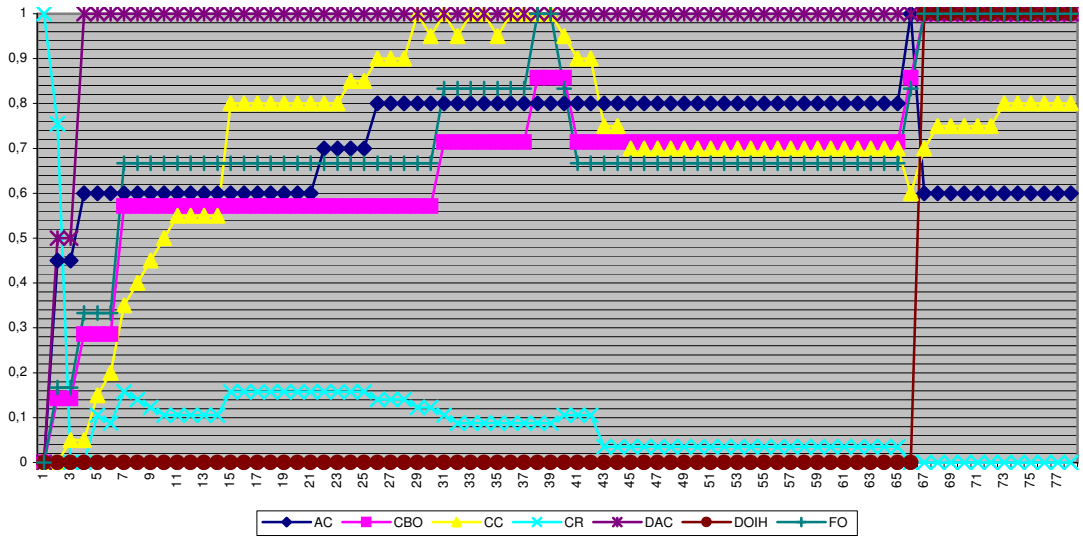
32. WMPC2 - *Weighted Methods Per Class 2*

Esta métrica tem o objetivo de medir a complexidade de uma classe, assumindo que uma classe com mais métodos que outra é mais complexa e que um método com mais parâmetros que outro também tem maior complexidade. Esta métrica conta os métodos e seus respectivos parâmetros para uma classe. Apenas métodos especificados em uma classe são considerados, sendo excluídos os métodos herdados.

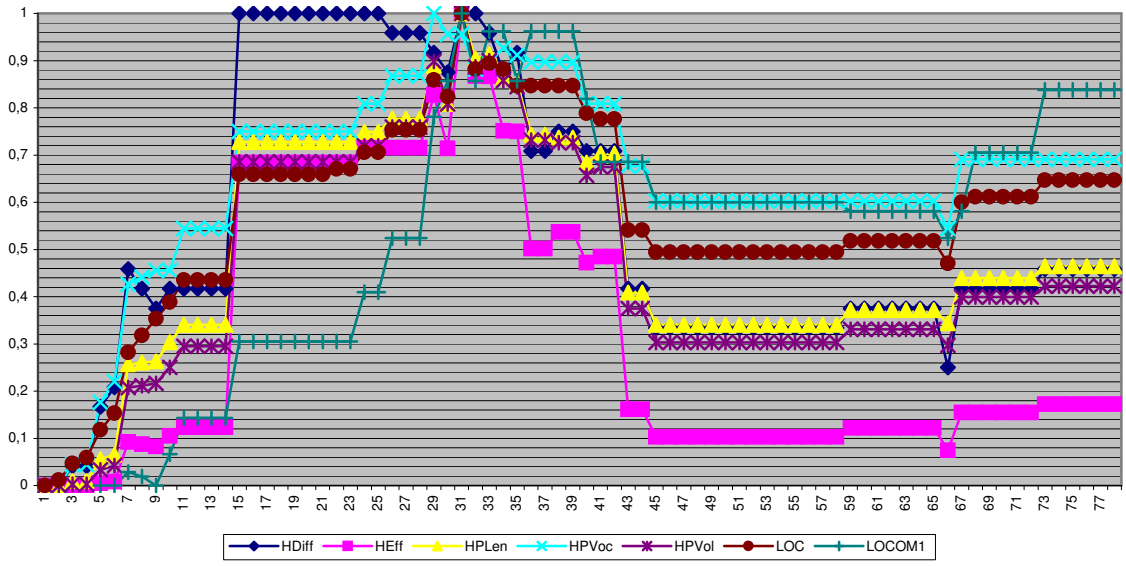
APÊNDICE 02 – MEDIÇÃO DA CLASSE *SITE* AO LONGO DA SEQUÊNCIA DE *REFACTORINGS*

Conforme relatado em (Lewerentz e outros, 2000), uma boa visualização dos valores obtidos das métricas melhora a eficiência no tratamento de grande quantidade de dados e na interpretação dos mesmos. Neste caso, foi adotado o diagrama de tendências para a apresentação das variações da classe *Site* ao longo da seqüência de *refactorings*. O Diagrama de Tendências é usado quando as mesmas métricas são calculadas para diferentes versões do mesmo sistema. A tendência dos valores para as entidades medidas ao longo das versões é de interesse para o objetivo desta dissertação. Em relação ao gráfico de tendências, tem-se a limitação de que as classes devem ter o mesmo nome em todas as versões. Toda reestruturação de classe que tenha como conseqüência divisão, agrupamento ou mudança de nome deve ser acompanhada e registrada para que seja possível considerar a mesma na visualização. Outra limitação é a conexão entre os valores das medições realizadas. A conexão linear é somente uma suposição, podendo não ser correta. O gráfico deverá somente considerar as versões que estão disponíveis, podendo ter existido outras versões entre a versão "n" e "n+1" (onde o estado do sistema tenha alterado de forma significativa).

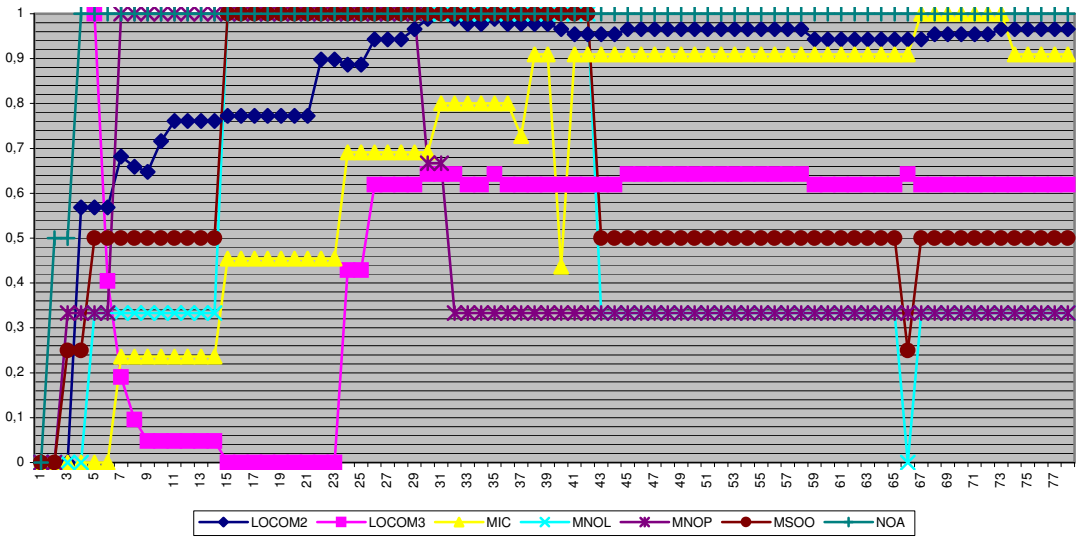
Tendência para Classe Site (Valores Normalizados)



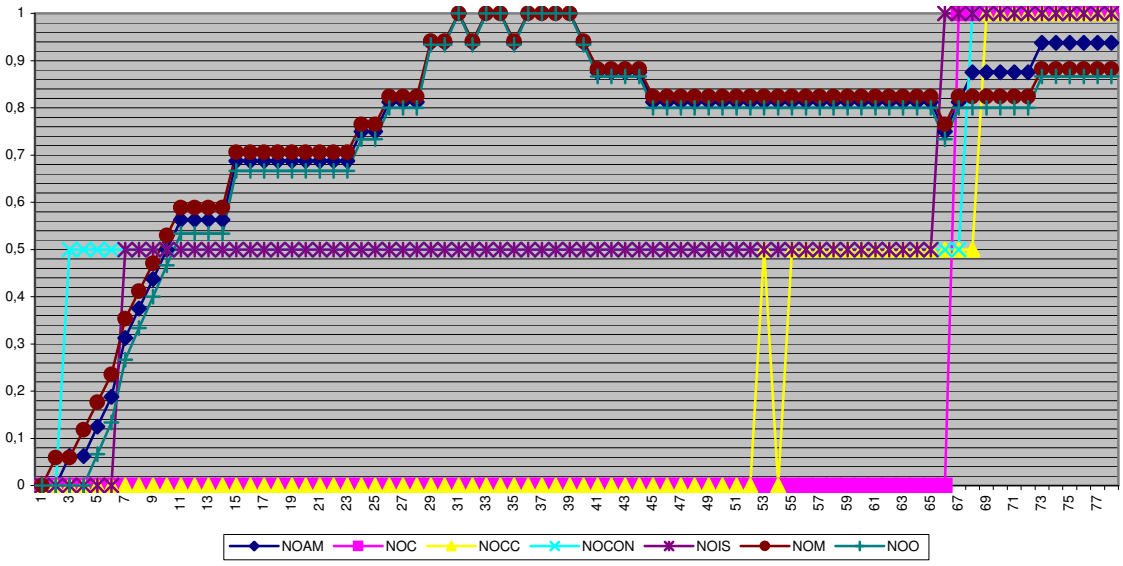
Tendência para Classe Site (Valores Normalizados)



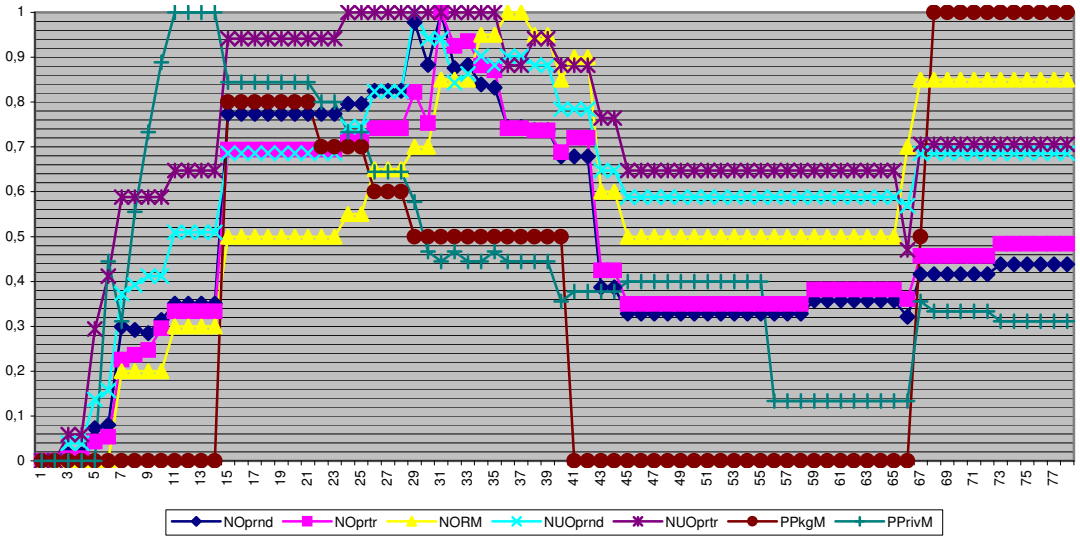
Tendência para Classe Site (Valores Normalizados)



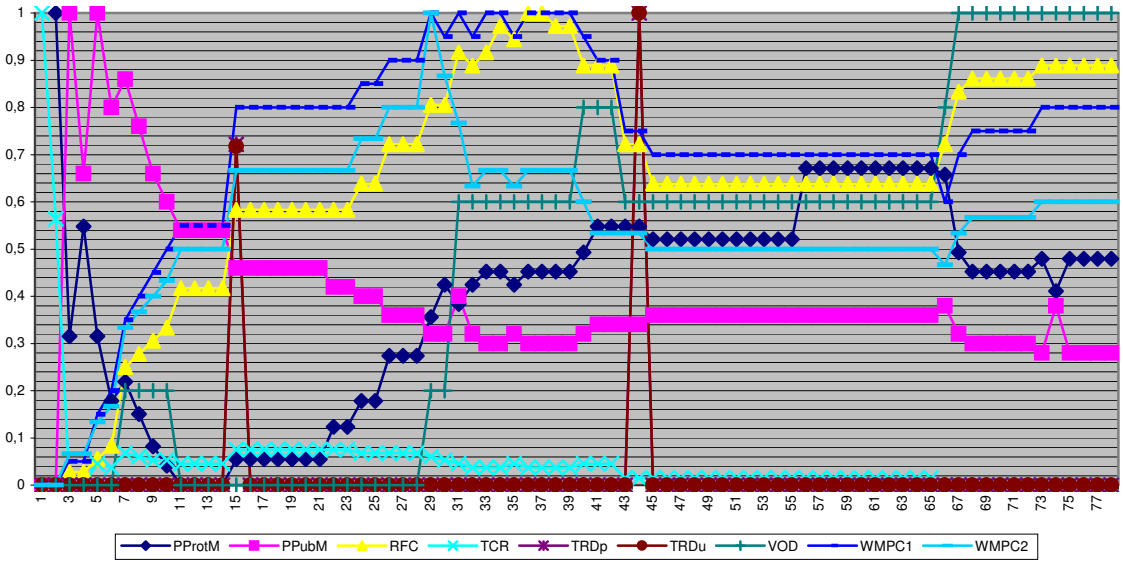
Tendência para Classe Site (Valores Normalizados)



Tendência para Classe Site (Valores Normalizados)



Tendência para Classe Site (Valores Normalizados)



APÊNDICE 03 – RELAÇÃO DE REFACTORINGS COM SUAS RESPECTIVAS DENOMINAÇÕES EM INGLÊS

- Migração de Trecho de Código para um Novo Método: *Extract Method*
- Dissolução de Método: *Inline Method*
- Dissolução de Variável Temporária: *Inline Temp*
- Substituição de Variável Temporária por Consulta: *Replace Temp with Querry*
- Introdução de Variável Esclarecedora: *Introduce Explaining Variable*
- Substituição de Variável Temporária por Duas ou Mais: *Split Temporary Variable*
- Uso de Variável Temporária: *Remove Assignments to Parameters*
- Substituição de Método por Objeto: *Replace Method with Method Object*
- Substituição por Algoritmo: *Substitute Algorithm*
- Migração de Método para Outra Classe: *Move Method*
- Migração de Variável para Outra Classe: *Move Field*
- Migração de Trecho de Código para uma Nova Classe: *Extract Class*
- Dissolução de Classe: *Inline Class*
- Delegação Oculta: *Hide Delegate*
- Remoção de Delegação: *Remove Midle Man*
- Introdução de Método Estrangeiro: *Introduce Foreign Method*
- Introdução de Extensão Local: *Introduce Local Extension*
- Encapsulamento de Informações: *Self Encapsulate Field*
- Criação de Objeto para Tratamento de Informações*: Replace Data Value with Object*
- Modificação para Objeto do Tipo Referência*: Change Value to Reference*
- Modificação para Objeto do Tipo Valor*: Change Reference to Value*
- Substituição de Arranjo para Objeto*: Replace Array with Object*
- Separação da Interface da Lógica do Negócio: *Duplicate Observed Data*
- Transformação de uma Associação Unidirecional para Bidirecional: *Change Unidirectional Association to Bidirectional*

- Transformação de uma Associação Bidirecional para Unidirecional*: Change Bidirectional Association to Unidirectional*
- Inclusão de uma Constante para Representar um Número*: Replace Magic Number with Symbolic Constant*
- Transformação de uma Variável Pública em Privada com Mecanismos de Acesso*: Encapsulate Field*
- Encapsulamento de uma Coleção*: Encapsulate Collection*
- Substituição de Registro por Classe de Dados: *Replace Record with Data Class*
- Substituição de Tipo ded Código por Classe: *Replace Type Code with Class*
- Substituição de Tipo de Código por Subclasse: *Replace Type Code with Subclasses*
- Substituição de Tipo de Código por Objeto do Tipo Estado/Estratégia: *Replace Type Code with State/Strategy*
- Substituição de Subclasses por Variável: *Replace Subclasses with Fields*
- Decomposição de Condicional: *Decompose Conditional*
- Consolidação de Expressão Condicional: *Consolidate Conditional Expression*
- Consolidação de Trechos Duplicados em Expressões Condicionais: *Consolidate Duplicate Conditional Fragments*
- Remoção de Variável de Controle: *Remove Control Flag*
- Substituição de Condicionais Encadeadas por Cláusulas: *Replace Nested Conditional with Guard Clauses*
- Substituição de Condicional por Polimorfismo: *Replace Conditional with Polymorphism*
- Uso de Objeto Nulo: *Introduce Null Object*
- Introdução de Afirmação: *Introduce Assertion*
- Renomeando Métodos: *Rename Method*
- Adição de Parâmetros: *Add Parameter*
- Remoção de Parâmetros: *Remove Parameter*
- Separação de Consulta do Modificador: *Separate Query from Modifier*

- Parametrização de Método: *Parameterize Method*
- Substituição de Parâmetros por Métodos: *Replace Parameter with Explicit Methods*
- Uso de Objeto para Obtenção de Informações: *Preserve Whole Object*
- Substituição de Parâmetro por Método: *Replace Parameter with Method*
- Substituição de Parâmetro por Objeto: *Introduce Parameter Object*
- Remoção de Método de Atribuição de Valores: *Remove Setting Method*
- Ocultação de Método em Relação a outras Classes: *Hide Method*
- Substituição de Construtor por Subclasse: *Replace Constructor with Factory Method*
- Encapsulando Projeção de Tipo: *Encapsulate Downcast*
- Substituição de Código de Erro por Exceção: *Replace Error Code with Exception*
- Substituição de Exceção por Teste: *Replace Exception with Test*
- Migração de Variável para Superclasse: *Pull Up Field*
- Migração de Método para Superclasse: *Pull Up Method*
- Migração de Construtor para Superclasse: *Pull Up Constructor Body*
- Migração de Método para Subclasse: *Push Down Method*
- Migração de Variável para Subclasse: *Push Down Field*
- Migração de Trecho de Código para uma Nova Subclasse: *Extract Subclass*
- Migração de Trecho de Código para uma Nova Interface: *Extract Interface*
- Junção de Superclasse e Subclasse: *Collapse Hierarchy*
- Migração de Trecho de Código para um Novo Método Modelo: *Form Template Method*
- Substituição de Herança por Delegação: *Replace Inheritance with Delegation*
- Substituição de Delegação por Herança: *Replace Delegation with Inheritance*

APÊNDICE 04 – DIAGRAMAS DE CLASSE AO LONGO DA SEQUÊNCIA DE *REFACTORINGS*

As seis figuras a seguir apresentam os diagramas de classe da aplicação ao longo da seqüência de *refactorings*, retratando a da criação de uma superclasse e quatro subclasses.

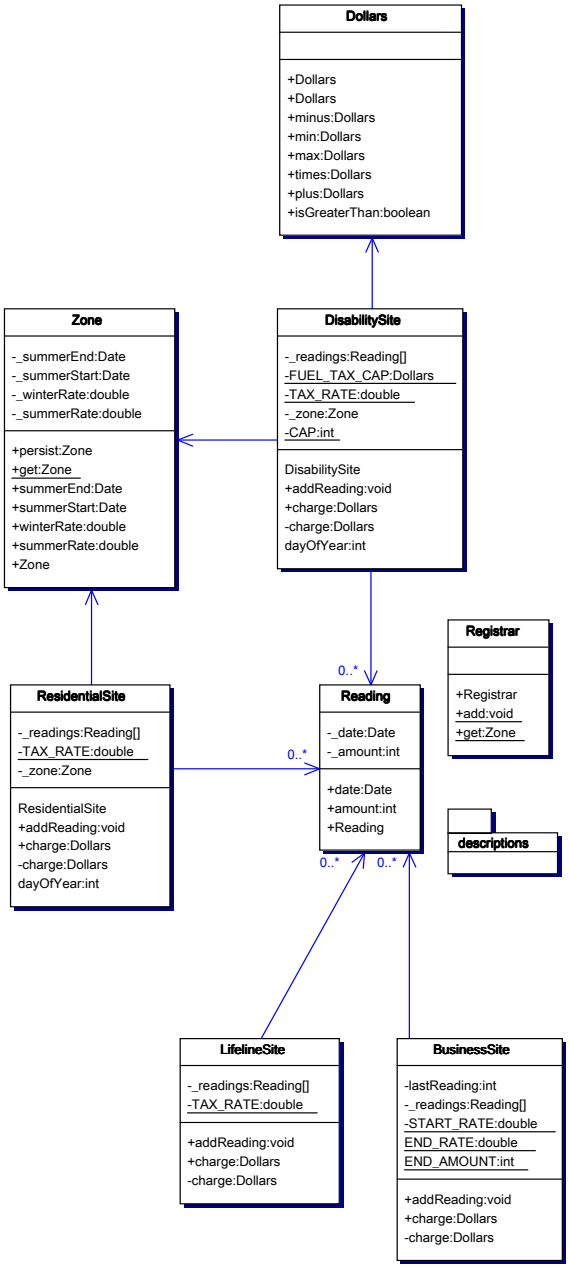


Figura A4.1: Aplicação Inicial

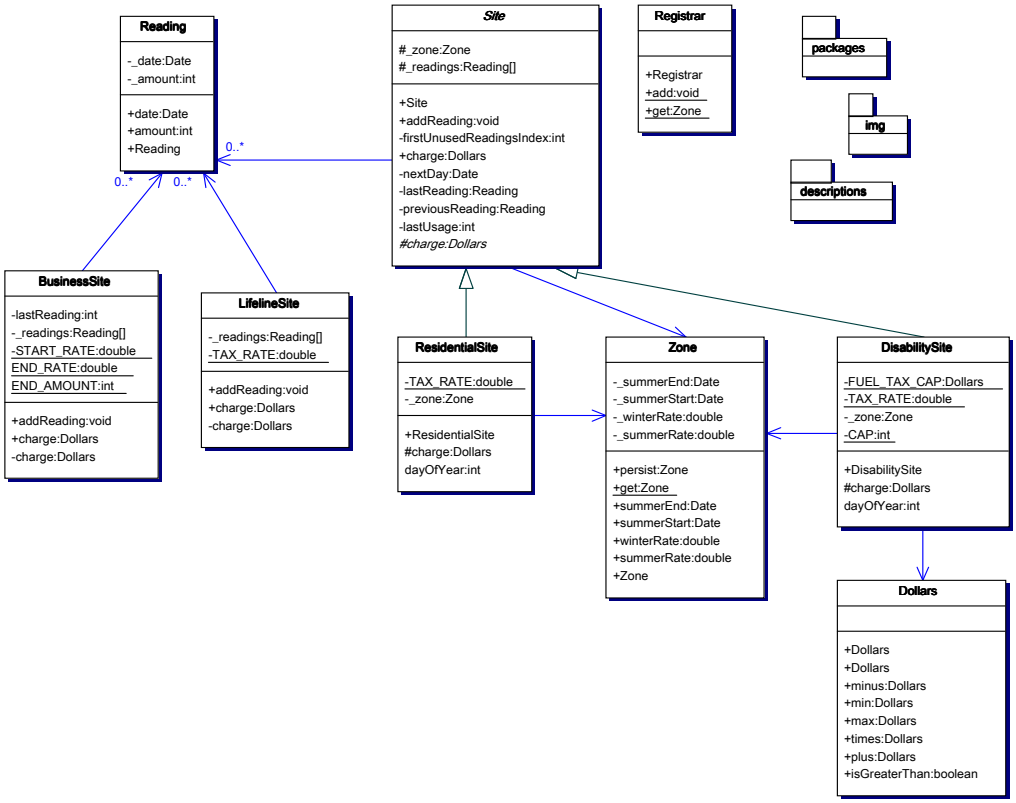


Figura A4.2: Aplicação após correção do *bad smells* 1 (Código Duplicado)

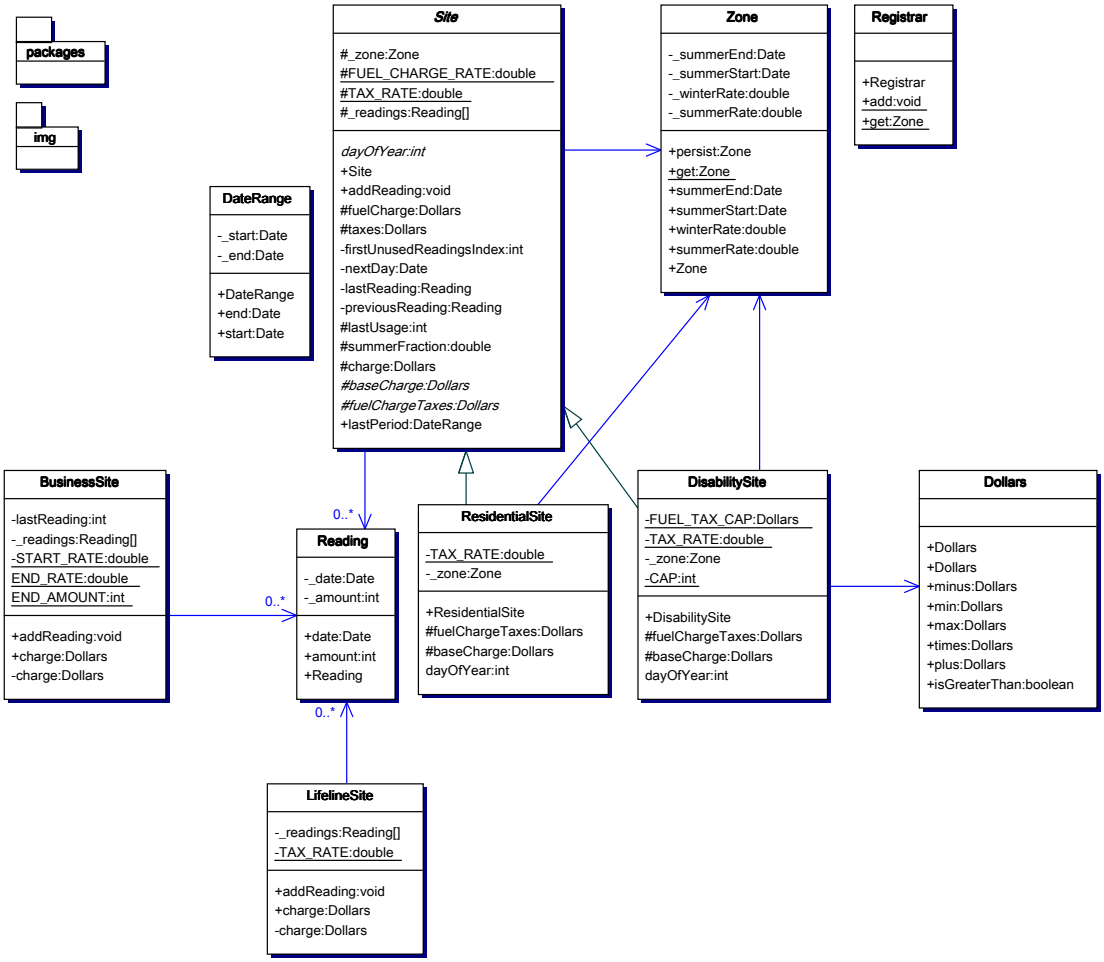


Figura A4.3: Aplicação após *bad smells* 2 (Método Longo/Código Duplicado)

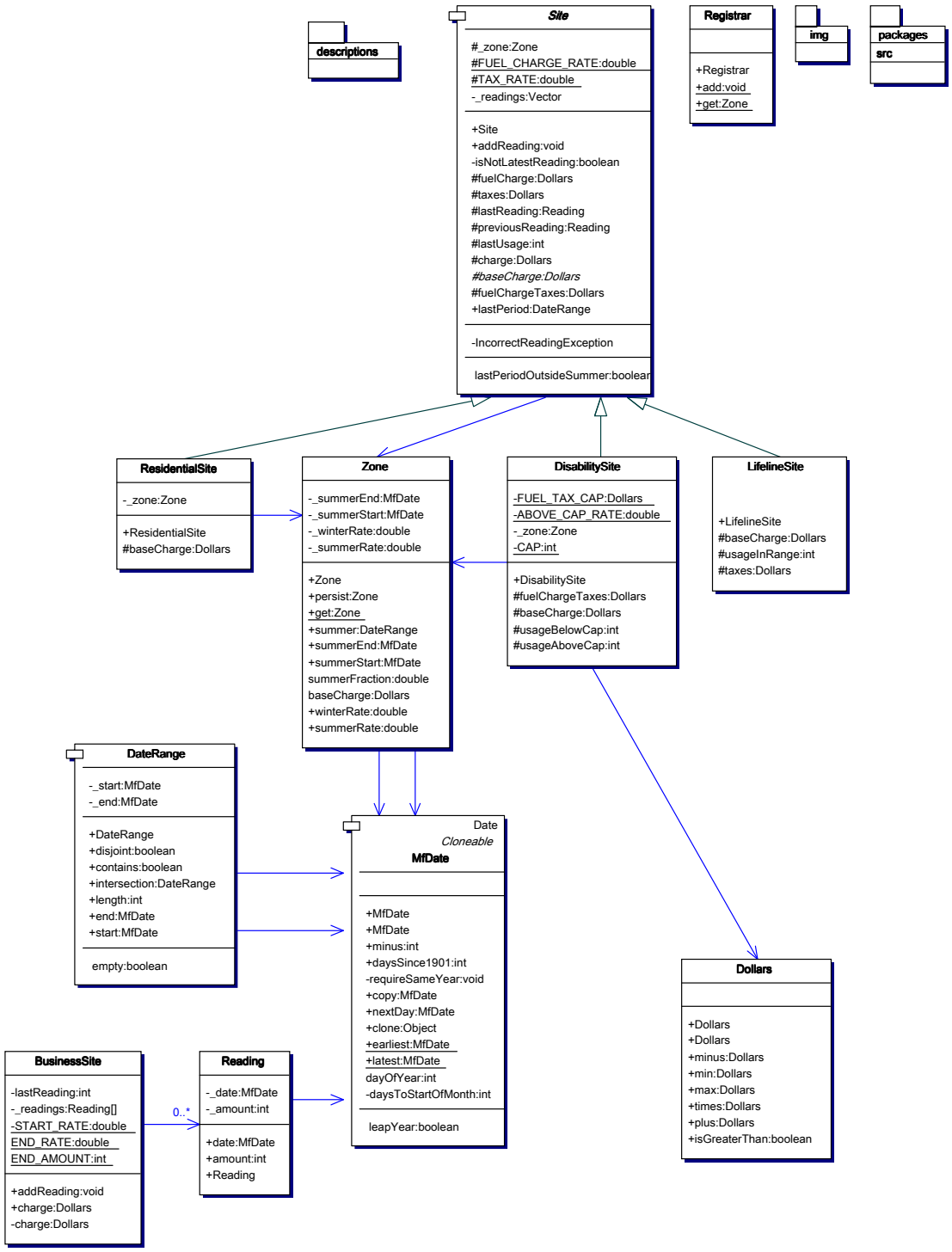


Figura A4.4: Aplicação após correção do bad smells 3 (Método Longo)

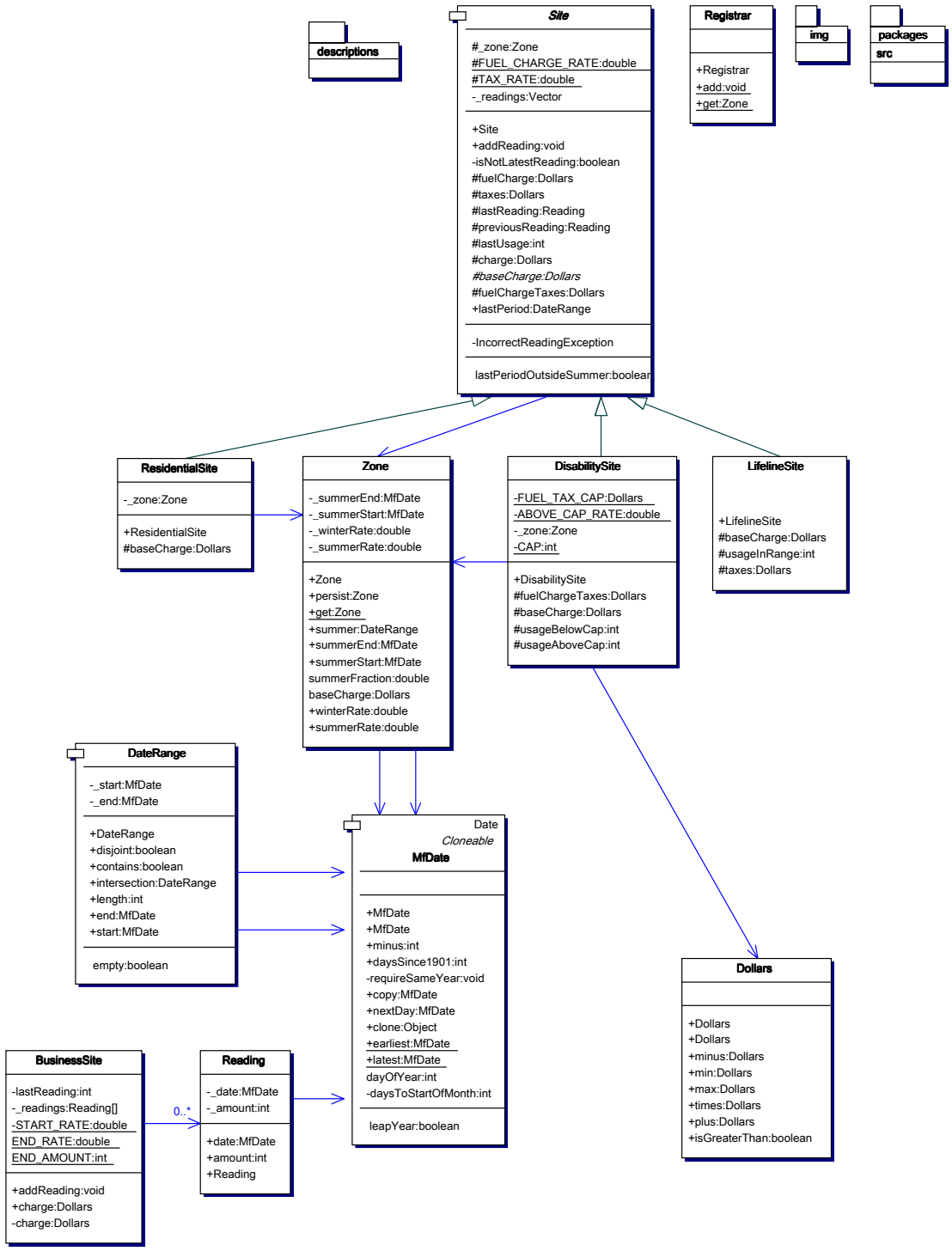


Figura A4.5: Aplicação após correção do *bad smells* 4 (Código Duplicado)

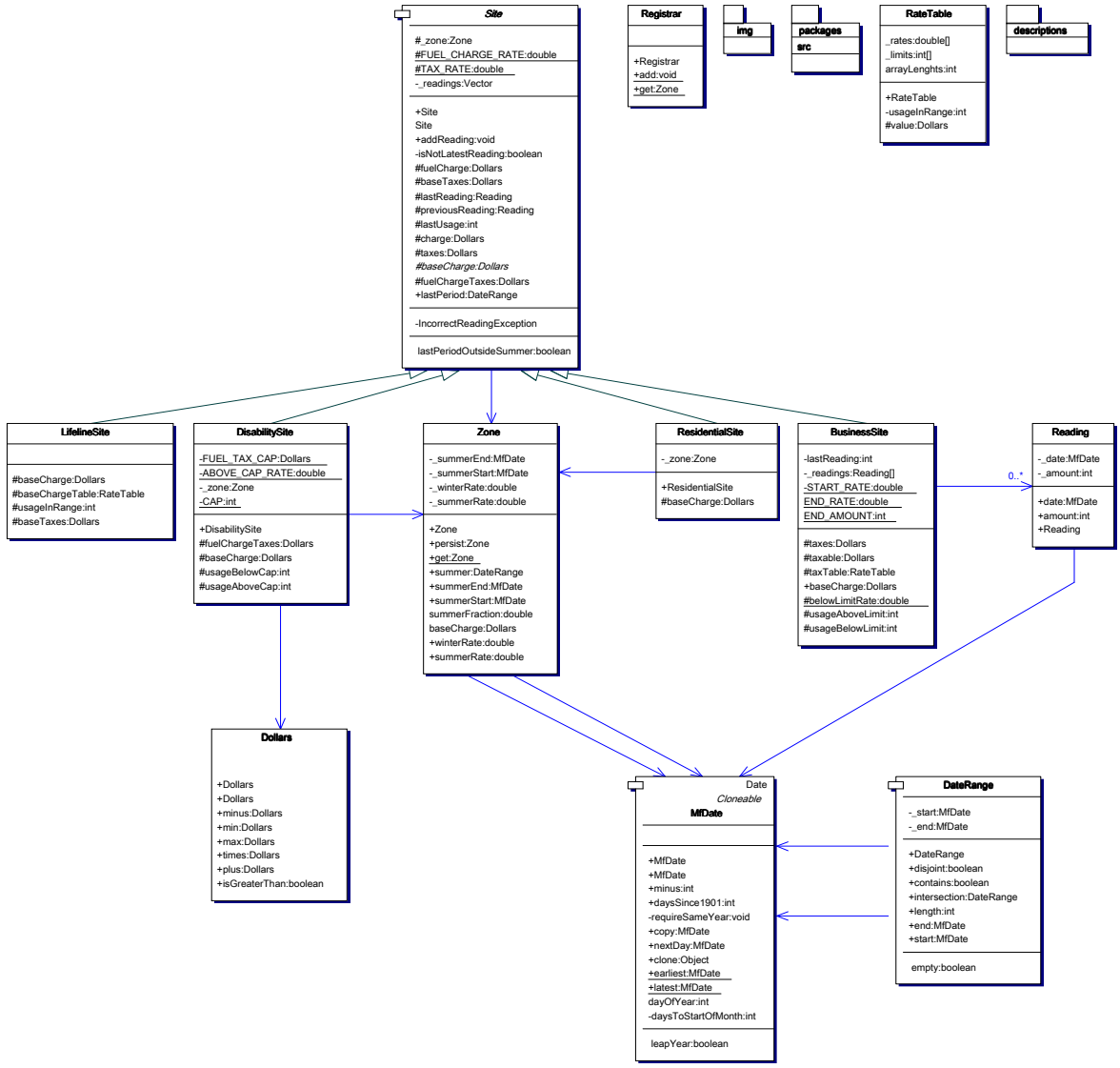


Figura A4.6: Aplicação após correção do *bad smells* 5