“SOCRATES – Sistema Orientado a objetos para CaRActerização de refaToraçõES”

André Toledo Piza de Moura

Trabalho Final de Mestrado Profissional em Computação

Instituto de Computação
Universidade Estadual de Campinas

"SOCRATES – Sistema Orientado a objetos para CaRActerização de refaToraçõES"

André Toledo Piza de Moura

24/02/2006

Banca Examinadora:

- **Prof. Dr. Marcos Lordello Chaim (Orientador)**
  Escola de Artes, Ciências e Humanidades – USP
- **Prof. Dra. Eliane Martins (Co-orientadora)**
  Instituto de Computação - Unicamp
- **Prof. Dr. Plínio Roberto Souza Vilela**
  Faculdade de Ciências Exatas e da Natureza - Unimep
- **Prof. Dra. Cecília Mary Fischer Rubira**
  Instituto de Computação – Unicamp
- **Prof. Dr. Edmundo Roberto Mauro Madeira**
  Instituto de Computação - Unicamp

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Júlia Milani Rodrigues – CRB8a / 2116

Moura, André Toledo Piza de

M865s "SOCRATES – Sistema Orientado a objetos para CaRActerização de refaToraçõES" / André Toledo Piza de Moura -- Campinas, [S.P. :s.n.], 2006.

Orientador : Marcos Lordello Chaim; Eliane Martins

Trabalho final (mestrado profissional) - Universidade Estadual de Campinas, Instituto de Computação.

1. Inteligência artificial. 2. Java (Linguagem de programação de computador). 3. Padrões de projeto . I. Chaim, Marcos Lordello. II. Martins, Eliane. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Título em inglês: "SOCRATES – Object Oriented System for Characterization of Refactorings

Palavras-chave em inglês (Keywords): 1. Artificial inteligence. 2. Java (Computer program language). 3. Design patterns.

Área de concentração: Engenharia da Computação

Titulação: Mestre em Engenharia da Computação

Banca examinadora: Prof. Dr. Marcos Lordello Chaim (EACH-USP)
Profa. Dra. Eliane Martins (IC-UNICAMP)
Prof. Dr. Plínio Roberto Souza Vilela (FCEN-UNIMEP)
Profa. Dra. Cecília Mary Fischer Rubira (IC-UNICAMP)
Prof. Dr. Edmundo Roberto Mauro Madeira (IC-UNICAMP)

Data da defesa: 24/02/2006

Programa de Pós-Graduação: Mestrado Profissional em Computação

"SOCRATES – Sistema Orientado a objetos para CaRActerização de refaToraçõES"

Este exemplar corresponde à redação final do Trabalho Final devidamente corrigido e defendido por André Toledo Piza de Moura e aprovado pela Banca Examinadora.

Campinas, 24/02/2006.

Prof. Dr. Marcos Lordello Chaim
(Orientador)

Prof. Dra. Eliane Martins
(Co-orientadora)

Trabalho Final apresentado ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Computação na área de Engenharia de Computação

TERMO DE APROVAÇÃO

Trabalho Final Escrito defendido e aprovado em 24 de fevereiro de 2006, pela Banca Examinadora composta pelos Professores Doutores:

Cecília Mary Fischer Rubira

Profa. Dra. Cecília Mary Fischer Rubira
IC - UNICAMP

Prof. Dr. Plínio Roberto Souza Vilela
UNIMEP

Prof. Dr. Marcos Lordello Chaim
EACH – USP

Dedico este trabalho a minha avó Nair Zaiden Toledo e mãe Rita H. Toledo Piza responsáveis e incentivadoras de toda a minha formação profissional e, muito mais importante, a formação do meu caráter. A Raquel S. Ribeiro pela paciência, compreensão e noites mal dormidas e a Maria, minha mais nova inspiração.

Agradecimentos

A Deus, por colocar as pessoas certas no meu caminho e me dar força para completar mais uma etapa na vida.

Ao professor Marcos L. Chaim, pela paciência, experiência, sabedoria e dedicação, que fizeram de um sonhador sem limites um pesquisador científico e empreendedor deste trabalho.

A todas as pessoas que de alguma forma participaram desta fase em minha vida, em especial: Gustavo e Alessandro, por facilitarem minha ida ao mestrado; Domingos, Krishna, Fabiane, Rogério e Spock, pelo estímulo e crescimento profissional; Jose Luis e Nemer, eternos amigos; Dinailton, pela companhia nas viagens entre Goiânia e Campinas; Bruna, Isabela e Toinho, pela força e apoio; e Fabio Jr, por ser um grande amigo.

Resumo

Refatoração é o ato de modificar o código fonte de um programa de computador sem, contudo, modificar seu comportamento observável. Em outras palavras, é alterar a estrutura de um sistema de software sem que isso provoque qualquer efeito no resultado final de suas funcionalidades. As modificações são feitas visando deixar o código mais fácil de ser entendido por terceiros que venham a modificá-lo e, conseqüentemente, diminuir os custos de sua manutenção. Entretanto, esta atividade é complexa e sujeita a erros, uma vez que normalmente é realizada de forma manual e depende das habilidades e da obediência a padrões do mantenedor que está analisando o código. Os pontos do software onde refatorações devem ser aplicadas constituem oportunidades de refatoração. A construção de sistemas automáticos para a detecção de oportunidades de refatoração requer a implementação de ambientes para análise de código e de tratamento regras para identificação de padrões no código fonte. Este trabalho apresenta o SOCRATES – Sistema Orientado a objetos para CaRacterização de refaToraçõES – cujo objetivo é fornecer auxílio automático para a identificação dos pontos candidatos a serem refatorados, isto é, oportunidades de refatoração. Para atingir este objetivo, SOCRATES utiliza uma arquitetura leve. Essa arquitetura é baseada em ferramentas livres e disponíveis e requer pouca codificação adicional. A codificação adicional é utilizada para que as ferramentas trabalhem em conjunto e para que os algoritmos de identificação das oportunidades de refatoração sejam implementados de forma eficiente. A presente versão do SOCRATES identifica de maneira automática a oportunidade de refatoração *parâmetro obsoleto* e mostra que os conceitos arquiteturais utilizados são válidos.

Abstract

Refactoring is the activity of modifying a computer program's source code without changing its external behavior. In other words, it consists of changing a software system's structure without affecting its functionalities. The changes are done in order to let the code more understandable for humans that might work on it. In this sense, its goal is to lower maintenance costs. Nevertheless, this activity is complex and error prone since it is usually carried out manually and is dependent on the maintainer's abilities and on his/her obedience to coding standards. The points of the software where refactorings should be applied are called refactoring opportunities. Building automatic systems to detect refactoring opportunities requires the implementation of environments which include source code analyzers and the treatment of rules to detect related patterns and standards. This work introduces SOCRATES – Sistema Orientado a objetos para CaRacterização de refaToraçõES (Object Oriented System for Characterization of Refactorings) – its main purpose is to automatically detect points to be refactored in the software, i.e., refactoring opportunities. To achieve this objective, SOCRATES is built upon a lightweight architecture. This architecture is based on open source tools and requires few additional coding. The additional coding was done to make the tools work together so that refactoring opportunities searcher algorithms could effectively work. The present version of SOCRATES identifies the *obsolete parameter* refactoring opportunity and shows that the architecture fundamentals used are valid.

Índice

CAPÍTULO 1 1

INTRODUÇÃO 1
1.1 Motivação 1
1.2 Contexto 3
1.3 Objetivos 4
1.4 Organização 4

CAPÍTULO 2 5

CONCEITOS BÁSICOS E TRABALHOS RELACIONADOS 5
2.1 Refatoração 5
2.1.2 Cheiros e Testes 7
2.1.3 Refatoração: manutenção preventiva 8
2.1.4 Quando aplicar uma refatoração? 9
2.1.5 Limitações 11
2.2 Meta-Programação e Reflexão 12
2.2.1 Histórico 12
2.2.2 Reflexão 13
2.2.3 Meta-programação e Java 14
2.3 Programação com regras em linguagem Java 15
2.3.1 Introdução 15
2.3.2 Máquina de Regras 17
2.3.3 Sistemas baseados em regras e a identificação de oportunidades de refatoração 19
2.4 Trabalhos relacionados 19
2.4.1 SOUL 19
2.4.2 Identificação de oportunidades de refatoração por meio de métricas 24
2.5 Considerações finais 27

CAPÍTULO 3 29

SOCRATES - SISTEMA ORIENTADO PARA CARACTERIZACAO DE REFATORAÇÕES 29
3.1 Motivação 29
3.2 A anatomia do SOCRATES 30
3.2 Seqüência de atividades do SOCRATES 41
3.3 Considerações Finais 42

CAPÍTULO 4 45

DETALHES DE IMPLEMENTAÇÃO 45
4.1 Componentes de infra-estrutura 45
4.1.1 Jikes 45
4.1.2 Digester 48

4.1.3 Drools 49
4.2 Componentes de negócio 52
4.2.1 Amarração XML (XML Binding) 53
4.2.2 Meta-Info 54
4.2.3 Análise 55
4.2.1. Regras de inferência 58
4.3 Inclusão de novas oportunidades de refatoração no SOCRATES 59
4.4 Alterando os Componentes de Negócio do SOCRATES 61
4.5 Exemplo de utilização do SOCRATES 62
4.5 Considerações Finais 65
CAPÍTULO 5 67
CONCLUSÕES 67
5.1 Síntese do Trabalho 67
5.2 Contribuições e Trabalhos Futuros 71
REFERÊNCIAS BIBLIOGRÁFICAS 73
APÊNDICE A 75
CÓDIGO FONTE DO PROGRAMA JAVA PARA PARÂMETRO OBSOLETO 75

Lista de Figuras

FIGURA 2.1 - APLICAÇÃO DA REFATORAÇÃO EXTRACT CLASS NA CLASSE PESSOA_A.....6
FIGURA 2.2 – EXEMPLO QUE ILUSTRA UM MECANISMO GENÉRICO DE REFLEXÃO.....13
FIGURA 2.3 – HIERARQUIA DE CLASSES.....21
FIGURA 3.1– ESTRUTURA DE COMPONENTES DO SOCRATES.....31
FIGURA 3.2 – FLUXO DE INFORMAÇÕES DOS COMPONENTES DO SOCRATES. 32
FIGURA 3.3 – TRECHO DE PROGRAMA JAVA.....32
FIGURA 3.4 – ARQUIVO JAVAML GERADO PELA FERRAMENTA JIKES 33
FIGURA 3.5 – CLASSE JAVA MAPEADA PELO COMPONENTE DIGESTER A PARTIR DO ARQUIVO JAVAML.....34
FIGURA 3.6 – CLASSE QUE REPRESENTA O ARQUIVO JAVA EM DISCO A SER ANALISADO. 35
FIGURA 3.7 – CLASSE QUE REPRESENTA AS META-INFORMAÇÕES DA CLASSE JAVA A SER ANALISADA.....36
FIGURA 3.8 – INSTANCIAÇÃO DA CLASSE JAVA CRIADA PELO COMPONENTE DIGESTER PARA UTILIZAÇÃO PELO SOCRATES. 37
FIGURA 3.9 – ARQUIVOS XML DE REGRAS PARA IDENTIFICAR O MAU CHEIRO PARÂMETRO OBSOLETO.....38
FIGURA 3.10 – REPRESENTAÇÃO ESTRUTURAL DO COMPONENTE DE ANÁLISE E SUA POSIÇÃO NA ESTRUTURA DE PACOTES COM TODOS OS COMPONENTES DO SOCRATES. 40
FIGURA 3.11 – SEQÜÊNCIA DE ATIVIDADES DO SOCRATES.....41
FIGURA 3.12 – RESULTADO DA ANÁLISE PELO SOCRATES PARA A IDENTIFICAÇÃO DE PARÂMETROS OBSOLETOS NO PROGRAMA DA FIGURA 3.1. 42
FIGURA 4.1 – PROGRAMA EXEMPLO [BAD00].....46
FIGURA 4.2 – ARQUIVO JAVAML GERADO PELO PROCESSO DE COMPILAÇÃO DO JIKES COM O MÉTODO XMLUNPARSE.....47
FIGURA 4.3 – FORMATO DA REGRA DENTRO DA MÁQUINA DE REGRAS DROOLS.....50
FIGURA 4.4 – CRIANDO UMA MEMÓRIA DE TRABALHO. 51
FIGURA 4.5 – INTERAGINDO COM A MEMÓRIA DE TRABALHO E INICIANDO A EXECUÇÃO DO MOTOR DE INFERÊNCIAS. 51
FIGURA 4.6 –ARQUIVO DE REGRAS XML DO COMPONENTE DROOLS PARA IDENTIFICAÇÃO DE PARÂMETROS OBSOLETOS.....56
FIGURA 4.7 –ALGORITMO UTILIZADO NA REGRA DE INFERÊNCIA PARA IDENTIFICAR PARÂMETROS OBSOLETOS.....59
FIGURA 4.8 – CLASSE MASTER.JAVA UTILIZADA PARA PROVA DE CONCEITO. 62
FIGURA 4.9 – RESULTADO DO PROCESSAMENTO REALIZADO PELO JIKES NO PROGRAMA DA FIGURA 4.8. 64
FIGURA 4.10 – RESULTADO DO PROCESSO DE ANÁLISE DO SOCRATES PARA A CLASSE MASTER.JAVA.....65

Lista de Tabelas

TABELA 2.1 – MAPAS DE REPRESENTAÇÃO DE PREDICADOS UTILIZADOS PELA FERRAMENTA SOUL [TOU03]. 22
TABELA 2.2 – POSSÍVEIS OCORRÊNCIAS DO MAU CHEIRO CÓDIGO DUPLICADO E SUGESTÕES DE REFATORAÇÃO [CAR03]. 25
TABELA 2.3 – PERGUNTAS E MÉTRICAS DERIVADAS PARA O MAU CHEIRO CÓDIGO DUPLICADO [CAR03]. 26
TABELA 4.1 – RELAÇÃO DE DEPENDÊNCIAS ENTRE OS COMPONENTES INTERNOS DO SOCRATES.....61

Capítulo 1

Introdução

1.1 Motivação

A atividade de manutenção tem importância fundamental no desenvolvimento de programas de computador. Estima-se que cerca de 60% do ciclo vida de um programa esteja relacionado às atividades de manutenção [May96]. A manutenção é tradicionalmente entendida como a fase do ciclo de vida que acontece após o software ter sido entregue ao cliente. Durante a manutenção, diferentes tarefas são realizadas como: correção de defeitos que foram descobertos durante a utilização em produção do sistema; migração do software para novas plataformas tecnológicas (e.g., uma nova interface, uma nova arquitetura etc.); inclusão de novas funcionalidades identificadas *a posteriori* pelos usuários; e reorganização da estrutura ou do código do software visando facilitar modificações futuras.

Assim, pode-se entender as tarefas de manutenção como qualquer mudança no software depois da distribuição ao cliente ou da colocação em operação, sejam essas mudanças evolutivas, corretivas, adaptativas ou preventivas [Pig97]. As mudanças evolutivas são também conhecidas como mudanças aperfeiçoadoras ou perfectivas. Elas são realizadas para satisfazer as novas necessidades do usuário. Já as mudanças corretivas consistem de correções no sistema causadas pela manifestação de defeitos do programa. As mudanças adaptativas, por sua vez, acontecem devido a requisitos específicos do ambiente em que o sistema irá executar. Por fim, as mudanças preventivas são mudanças que acontecem sem que defeitos sejam detectados. O objetivo da modificação é melhorar a estrutura ou o código do software de forma que mudanças futuras sejam realizadas com maior facilidade e sem que novos defeitos sejam inseridos; em outras palavras, visa facilitar futuras manutenções do software.

Os conceitos acima foram desenvolvidos considerando atividades que ocorrem depois do software ter sido entregue. No entanto, com advento dos métodos ágeis de desenvolvimento de software [Bec99], eles foram reavaliados em um novo contexto. Atividades realizadas apenas depois da entrega do software agora ocorrem dentro do processo de desenvolvimento, possibilitando economia de prazos e custos, com qualidade, eventualmente, superior à dos métodos tradicionais. Por exemplo, a realização de mudanças que visam melhorar as estrutura do software é uma das práticas dos métodos ágeis, em especial da chamada Programação Extrema [Bec99]. Esta é uma atividade típica de manutenção que em Programação Extrema é realizada no desenvolvimento.

Assim, a atividade de manutenção vem ganhando nova força, aumentando sua importância dentre as disciplinas de engenharia de software. Nessas novas formas de desenvolvimento (métodos ágeis), o software deve ser entregue ao cliente tão logo já possua funcionalidade suficiente [Bec99]. Pode-se entender as diversas versões intermediárias do software (*releases*) como versões do sistema que sofrem mudanças evolutivas até incluir todas as funcionalidades desejadas pelo cliente. O pressuposto é que o custo de mudança é constante no desenvolvimento baseado em métodos ágeis. Portanto, neste contexto, conceitos de manutenção de software são utilizados antes e depois da entrega do software.

Do tempo gasto em manutenção, estima-se que cerca de 50% seja gasto compreendendo o código fonte para se descobrir quais as modificações são necessárias [Pig97]. Os pontos de modificação nem sempre são fáceis de ser encontrados e, às vezes, é necessário um estudo profundo do software para poder identificar uma pequena necessidade de modificação. Mesmo que o mantenedor conheça bem o trecho de código em que irá trabalhar, como a análise do código é uma atividade essencialmente manual, ela é muito suscetível a enganos e problemas de mau entendimento.

Existe ainda um outro agravante: se o mantenedor não conhecer o código, ao tentar modificá-lo, a chance de introduzir novos defeitos é muito grande, já que pode haver

conseqüências e efeitos inesperados no código manipulado. Portanto, uma questão relevante em qualquer atividade de manutenção é identificar os pontos nos quais modificações são necessárias. Supondo que o mantenedor consiga encontrar e isolar o problema, um menor tempo é gasto na aplicação de modificações [Pig97].

1.2 Contexto

Para as modificações que visam melhorar a estrutura e o código do sistema – manutenções preventivas, existem técnicas e ferramentas que auxiliam a tarefa de aplicação destas modificações. Grande parte destas técnicas estão reunidas sob o conceito de *reestruturação/refatoração* [Men04]. *Reestruturação* são mudanças feitas na estrutura do software no sentido de deixá-lo mais fácil de ser entendido e menos custoso de ser modificado, sem, contudo, modificar seu comportamento observável. O termo *refatoração* foi cunhado por Opydike [Op92 apud Men04] como "o processo de modificação de um sistema (orientado a objeto) de software de forma que o sistema não tenha o seu comportamento externo alterado, embora a sua estrutura interna seja melhorada".

Segundo Mens e Touwé [Men04], *refatoração* envolve o mesmo conceito de reestruturação, porém, aplicado ao contexto dos sistemas orientados a objetos, sendo a idéia central redistribuir classes, variáveis e métodos na hierarquia de classes a fim de facilitar futuras adaptações extensões. Fowler e Beck [Fow99] referem-se às "estruturas no código que sugerem a (às vezes gritam pela) possibilidade de refatoração" como *maus cheiros*. Este termo inusitado[1] visa indicar pontos do software que são candidatos naturais à aplicação de refatorações. Neste sentido, os maus cheiros constituem *oportunidades* para aplicação de refatorações.

Este trabalho apresenta um sistema automático para detecção de maus cheiros, isto é, para detectar oportunidades de refatoração. Dessa forma, está-se automatizando uma parte custosa da atividade de manutenção – e também de desenvolvimento, no caso dos

[1] O termo *mau cheiro* tem origem no cuidado com bebês pequenos – se está com mau cheiro, então deve-se trocar a frauda. No caso do software, se o código cheira mal, deve ser modificado.

métodos ágeis – que é a compreensão do software; especificamente, a compreensão do código para identificação de maus cheiros e posterior aplicação de refatorações.

1.3 Objetivos

Este trabalho final tem como objetivo apresentar um sistema automático de detecção de oportunidade de refatorações. Além desse objetivo principal, o sistema proposto possui os seguintes sub-objetivos:

1. Utilizar ferramentas livres para reduzir a necessidade de programação não relacionada com a detecção de oportunidades de refatoração;
2. Permitir a implementação de algoritmos eficientes.

.

A idéia é implementar uma abordagem leve que utilize ferramentas disponíveis para reduzir a programação e que permita a implementação eficiente de algoritmos para a detecção de oportunidades de refatoração.

1.4 Organização

Neste capítulo foram apresentados a motivação, o contexto e os objetivos deste trabalho final. No próximo capítulo são apresentados os conceitos básicos utilizados, bem como dois trabalhos relacionados. O Capítulo 3 apresenta, por meio de um exemplo simplificado, o SOCRATES – Sistema Orientado a objetos para CaRActerização de refaToraçõES – que implementa detecção automática de oportunidades de refatoração. O Capítulo 4 descreve os detalhes de implementação do SOCRATES, bem como um exemplo de utilização do sistema em uma de suas classes. No Capítulo 6, as conclusões são apresentadas.

Capítulo 2

Conceitos básicos e trabalhos relacionados

Neste capítulo são discutidos os conceitos básicos utilizados no SOCRATES (Sistema Orientado para CaRActerizacao de refaToraçõES). Inicialmente, apresentam-se os conceitos fundamentais de *refatoração* e *mau cheiro*. Em seguida, são apresentados os demais conceitos utilizados para identificar oportunidades de refatoração, a saber, meta-programação, reflexão e programação com regras em Java. São ainda descritos dois trabalhos relacionados com o SOCRATES.

2.1 Refatoração

Segundo Fowler e colegas [Fow99], a refatoração de um software pode ser entendida como "uma mudança interna na estrutura do software a fim de tornar mais fácil o seu entendimento e a sua modificação mais barata, sem, contudo, modificar seu comportamento observável". Uma refatoração é, normalmente, uma pequena mudança no software. Por exemplo, suponha-se que uma classe realiza o trabalho de duas; uma mudança que facilita o entendimento do código e a sua possível modificação futura é alterar o código de forma que cada classe seja responsável por um único trabalho, ou seja, tenha coesão funcional.

A Figura 4.1 apresenta a classe `Pessoa_A` que ilustra esta situação. A classe `Pessoa_A` possui atributos e métodos (e.g., `telefoneEscritório`, `getTelefoneParticular()`) relacionados com o número de telefone alocado a uma pessoa, além dos próprios atributos e métodos (e.g, `nome`, `getNome()`) da pessoa. A classe `Pessoa_A` pode ser dividida e dar origem a duas outras classes: `Pessoa_B` e `NumeroTelefone`. A nova configuração define mais claramente as funções de cada classe,

facilitando a manutenção e reuso.

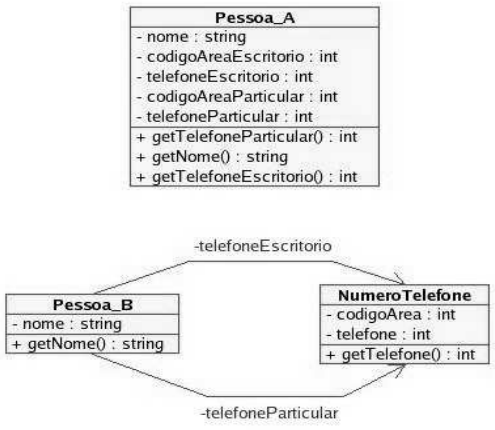

Figura 2.1 - Aplicação da refatoração extract class na classe Pessoa_A.

A refatoração que realiza este mudança é, segundo a definição de Fowler e colegas [Fow99], a refatoração *extract class*[2]. É importante observar que em geral uma refatoração provoca a ocorrência de outras refatorações. No exemplo da refatoração *extract class*, esta mudança, em geral, envolve a refatoração *extract method* (retira o método de uma classe e passa para outra) e a refatoração *extract field* (retira o campo de uma classe e passa para outra). Fowler e colegas definiram um catálogo de refatorações no qual são discorridos o objetivo e as modificações padrões que respeitam os propósitos de cada refatoração descrita no catálogo.

As subseções que se seguem apresentam vários aspectos da aplicação de refatorações e foram baseadas no texto de Fowler e colegas [Fow99].

[2] Optou-se por manter neste texto o nome das refatorações originais (em Inglês), pois traduções brasileiras de livros sobre conceitos de orientação a objeto têm adotado esta solução [Gam00]. No entanto, alguns autores [Car03] têm optado por traduzir os nomes das refatorações; quando o trabalho desses autores é citado a tradução por eles proposta foi utilizada.

2.1.2 Cheiros e Testes

O conceito de refatoração não é suficiente para identificar os pontos do software que precisam de melhorias. Para isso, é necessário cheirar o código de tal forma que, nos pontos onde o cheiro não estiver bom (*bad smells*), deve-se aplicar a refatoração adequada. O problema é que esta técnica pode ser muito subjetiva e variar entre projetos. Para isso, é preciso utilizar heurísticas e um conjunto de experiências anteriores que podem ajudar a correta identificação dos cheiros.

Uma lista dos maus cheiros mais comuns no desenvolvimento de software orientado a objetos foi elaborada [Fow99]. Dentre eles, podem-se destacar a *duplicação de código*, *método longo*, *classes muito longas*, os quais são problemas de fácil identificação. Porém, há outros que requerem maior análise e conhecimento da estrutura do software. Por exemplo, dados sempre encontrados em conjunto ("Data Clumps"), classes que parecem mais interessadas na lógica de outras classes que em sua própria lógica interna ("Feature Envy") e hierarquia de herança em paralelo, dentre outros.

As refatorações corrigem os problemas associados aos maus cheiros. Por exemplo: o problema de métodos longos quase sempre pode ser corrigido com o uso da refatoração *extract method* (busca partes do método que parecem ter um relacionamento mais forte e constrói outro método); porém, se este método for complexo e contiver muitas varáveis temporárias pode ser necessário aplicar conjuntamente a refatoração *replace temp with query* (elimina as variáveis temporárias). Esta situação é identificada como um mau cheiro porque as boas práticas de desenvolvimento e experiências anteriores mostram que métodos menores e com menos variáveis temporárias são mais fáceis de serem lidos e não degradam o desempenho de softwares orientados a objeto. O processo de identificação de maus cheiros que podem ser solucionados pela aplicação de uma ou mais refatorações é chamado de *identificação de oportunidades de refatoração*.

Outra questão importante na aplicação de refatorações é validar as transformações realizadas. Para isto, a pré-condição essencial é ter um conjunto sólido de casos de testes.

Mesmo com o uso de ferramentas automatizadas de apoio à refatoração, esta é uma tarefa que requer atenção especial. Isto porque atualmente as ferramentas disponíveis cobrem poucos casos de aplicação de refatorações e a maioria delas é aplicada manualmente.

Para apoiar a automação dos testes de validação da aplicação de uma refatoração é utilizado a biblioteca JUnit [Gam05]. O JUnit é uma interface de aplicação de programas (API – *Aplication Program Interface*) para a linguagem Java que se casa muito bem com a prática de refatoração. Basicamente, essa API utiliza o padrão *composite* [Gam00] que permite o agrupamento dos testes em conjuntos (*suites*). Estes conjuntos de testes podem ser executados automaticamente.

Portanto, o JUnit permite que testes sejam executados automaticamente e as saídas obtidas sejam automaticamente verificadas de tal forma que, se houver uma diferença entre a saída esperada e a obtida, o desenvolvedor é informado. Assim, depois da aplicação de refatoração em uma classe, deve-se executar os casos de teste automaticamente como o JUnit; caso haja diferença entre as saídas obtidas, então a refatoração aplicada não foi bem sucedida. Deve-se observar que é fundamental nesse contexto que os testes sejam bem projetados para validar as funcionalidades esperadas do software, pois são eles que vão garantir que as refatorações aplicadas não introduzem defeitos e não alteram a funcionalidade do trecho de código alterado.

2.1.3 Refatoração: manutenção preventiva

O principal propósito da refatoração é tornar o código mais compreensível por outros programadores. Das várias modificações que o software pode sofrer, apenas as que possuem este propósito podem ser consideradas refatorações. Se, por ventura, o desenvolvedor acrescentar qualquer tipo de funcionalidade, ou mesmo retirar alguma, estaremos diante de uma manutenção aperfeiçoadora ou corretiva.

O oposto da refatoração é a otimização de código, pois as mudanças em suas estruturas possuem propósitos contrários à sua legibilidade. Normalmente, o código

resultante deste tipo de operação fica mais difícil de ser entendido, devido à própria natureza das modificações almejarem um nível de código mais próximo do código de máquina ou operações muito aprimoradas, como é o caso do deslocamento de bits.

2.1.4 Quando aplicar uma refatoração?

É difícil definir qual seria o momento ideal para se aplicar refatoração. Se durante ou depois do desenvolvimento de um módulo do sistema ou, até mesmo, depois de todo o desenvolvimento estar concluído. No entanto, parece haver um consenso que o ideal seria durante o desenvolvimento.

Isto pode parecer contraditório com a afirmação de que a aplicação de uma refatoração é uma atividade de manutenção preventiva. Porém, em metodologias ágeis como Programação Extrema [Bec99], o desenvolvedor assume dois papéis diferentes no ciclo de desenvolvimento. Por exemplo, primeiro ele adiciona funcionalidade ao módulo; o módulo é testado e concluído, o que significa dizer que executa de forma correta. Em seguida, o desenvolvedor aplica refatorações a fim de deixar o código mais legível e manutenível; porém, o mesmo conjunto de testes executado inicialmente deve ser executado novamente (teste de regressão) sem apresentar falhas.

Portanto, tudo o que funcionava antes deve continuar funcionando de tal forma que o usuário não percebe que o código foi refatorado. Porém, um desenvolvedor deve ser capaz de entender o novo código em tempo muito menor e deixá-lo pronto para novas modificações de forma muito mais fácil.

O fundamento da refatoração baseia-se no fato de que um software é desenvolvido não apenas para que funcione hoje, mas para que os desenvolvedores no futuro possam fazê-lo funcionar com menos esforço, seja para entendê-lo, seja para modificá-lo. Em outras palavras, deseja-se que o mantenedor do código encontre as condições mais favoráveis possíveis para aumentar o ciclo de vida do software. Fowler e colegas elencam os motivos abaixo para aplicar refatorações:

- Refatoração melhora o projeto do código. De acordo com a lei do aumento da complexidade do software de Lehman [Pig97], a estrutura do código tende a degradar com sucessivas manutenções. As mudanças no código fonte são realizadas sem a total compreensão de seu projeto, fazendo com que perca sua estrutura original. E isto tem um efeito cumulativo, ou seja, fica cada vez mais difícil realizar a manutenção do código. Refatoração tem a capacidade de inverter esta regra.

- Refatoração faz o software mais fácil de ser entendido. O código fica mais familiar devido ao reaproveitamento das soluções e, portanto, mais claro e lógico a qualquer desenvolvedor. Segundo Fowler e colegas, qualquer programador consegue escrever código que uma máquina possa entender, mas somente programadores experientes e com o uso de ferramentas corretas conseguem fazer código que outras pessoas conseguem entender.

- Refatoração evita a introdução de defeitos. Os hábitos invocados pela aplicação de refatorações ajudam o desenvolvedor a codificar de forma mais robusta e efetiva. É difícil ser um excelente desenvolvedor, mas possuir bons hábitos ajuda bastante.

- Refatoração ajuda a programar mais rápido. No fim, todos os pontos anteriores justificam este. Isto pode não ser intuitivo, já que primar pela qualidade representa tempo gasto em estruturas não funcionais. Porém, o fundamento de um rápido desenvolvimento é a manutenção de uma boa estrutura de projeto que vai evitar ou tornar mais rápidas modificações futuras.

2.1.5 Limitações

A principal limitação das refatorações está no fato de ser uma tecnologia nova e cujos efeitos colaterais ainda não serem largamente conhecidos. Alguns dos problemas conhecidos são descritos abaixo [Fow99].

- Banco de dados. Em sistemas objeto-relacional há problemas devido ao forte acoplamento entre o banco de dados e as aplicações que o modelam, além disso, existe a dificuldade da migração de dados. Em sistemas de banco de dados não orientados a objetos este problema pode ser contornado com aplicação de camadas que isolam o banco da aplicação. Em sistemas puramente orientados a objetos a situação tende a ser mais homogênea e automática.

- Mudanças de interfaces. Mudar a implementação das interfaces é um processo que não tende a espalhar muitos problemas, já que cada objeto pode implementar uma interface comum da forma que lhe convier. Este tipo de mudança é inerente à orientação a objetos. Porém, mudar a interface é um processo que pode gerar efeitos colaterais imprevisíveis se esta interface estiver sendo usada por código que não pode ser encontrado e mudado, como é o caso de componentes fechados.

- Em alguns casos, o código está tão degradado que a aplicação de refatorações se torna ineficiente e, até mesmo, prejudicial. Nestes casos, o melhor é re-escrever o código do início. A melhor forma de certificar-se se esta é ou não a melhor decisão é aplicar testes. Se comprovado que o código não faz o que deveria, então, refatorá-lo em nada adiantará. Se ainda assim, a decisão continuar difícil, um bom caminho a ser tomado é divisão do código em blocos de forte acoplamento e tentar aplicar refatorações de forma a avaliar se o melhor é reconstruir os blocos ou refatorá-los. Este, porém, é um campo que necessita de maior esforço de pesquisa, pois

atualmente é realizado essencialmente baseado sentimento (“feeling”) das pessoas envolvidas.

2.2 Meta-Programação e Reflexão

2.2.1 Histórico

Em 1946 John Von Neuman introduziu o conceito de programas armazenados que define uma arquitetura abstrata composta de UCP (Unidade Central de Processamento) e memória. Assim, o programa fica “armazenado” disponível para consultas. Entretanto, com a tecnologia disponível em 1946, a representação desta informação em memória era muito complexa e confusa para a mente humana, pois o que se consultava era o programa em linguagem de máquina.

No final dos anos 50, surgiu a linguagem LISP (LISt Processing) que já incluia facilidades para manipular o código do programa. LISP é uma linguagem capaz de interpretar o código em formato ASCII. Para isto, o código fonte é mapeado no código binário através de um formato intermediário. Infelizmente, a tecnologia desta época ainda não permitia que esta linguagem chegasse a um mapeamento perfeito e nem todas instruções no código fonte possuem elementos correspondentes diretos no código compilado.

Mesmo assim, LISP é uma linguagem eficiente e seu interpretador consegue, dinamicamente, descobrir informações relevantes sobre as estruturas do código fonte em execução. Ela pode ser considerada uma evolução da máquina de Von Neuman e se tornou a primeira aplicação real do conceito de meta-Programação [Doe04]. Linguagens que incluem o conceito de meta-programação permitem buscar informações relativas à estrutura (e.g., variáveis declaradas, assinatura) e à arquitetura (hierarquia de classe) ou obter métricas sobre um componente ou fragmento de programa, em tempo de execução [McM97].

Considere-se um programa de calcular, por exemplo. A não é utilizada para saber quais operações matemáticas (adição, subtração, porcentagem) que ele é capaz de realizar, mas como as operações implementadas foram codificadas e conseguem cumprir seus objetivos. Em outras palavras, se a calculadora consegue somar dois números, deseja-se saber como ela, internamente, utiliza suas estruturas para realizar esta operação. Para isto, as consultas de meta-programação buscam informações sobre os nomes e tipos das classes, métodos e argumentos utilizados. A grosso modo, estas operações são semelhantes a consultas que buscam informações sobre tabelas e colunas em um banco de dados.

2.2.2 Reflexão

No final da década de 70, com o advento da linguagem Smalltalk [Gol89], foram introduzidas na arquitetura das linguagens de programação alguns objetos especiais capazes de guardar, em tempo de execução, informações de meta-programação sobre as classes desenvolvidas e ativar estas informações quando necessário.

Figura 2.2 – Exemplo que ilustra um mecanismo genérico de reflexão.

Observe-se a Figura 2.2, quando o programador está instanciando uma classe de sua regra de negócio, ele também estará criando um objeto interno na estrutura da linguagem que guarda informações sobre a classe, seus métodos e variáveis membro. Essas

informações podem ser acessadas e ativadas separadamente da classe a que se referem.

Em Java, a estrutura de reflexão apóia-se na classe `java.lang.Class` [Gre05]. Esta classe é criada automaticamente pela arquitetura da linguagem Java no momento em que o programador desenvolve a classe de regra de negócio da aplicação. A arquitetura de meta-programação da linguagem Java consegue alcançar as informações: nome/tipo de retorno dos métodos, tipo dos argumentos e nome/tipo das classes e variáveis membro; e, desta forma, alimentam a classe `java.lang.Class`.

Uma vez que as informações estão disponíveis, a linguagem Java é capaz de realizar programação por reflexão, ou seja, o programa é capaz de consultar informações por meta-programação e instanciar as estruturas obtidas em tempo de execução.

Em outras palavras, o código é capaz de investigar suas próprias definições, ativar fragmentos de código que não estão em uso e até mesmo modificar algumas estruturas dinâmicamente.

2.2.3 Meta-programação e Java

A linguagem de programação Java incluiu o conceito de meta-programação desde sua versão inicial, contando com recursos que são capazes de fazer um mapeamento entre o código fonte e o código compilado. Ainda assim, existem limitações importantes em Java que obrigam, nesta pesquisa, o uso de uma tecnologia híbrida.

A linguagem Java, através de meta-programação, não consegue, na versão 1.4 (utilizada no início desta pesquisa), inferir consultas em um nível granular fino. De forma que não é possível consultar informações internas a um método ou, ainda, saber quais os nomes de seus argumentos.

Este tipo de informação é essencial para a identificação de oportunidades de refatoração. Por este motivo, faz-se necessário o uso de um artefato tecnológico externo a

arquitetura da linguagem Java capaz de gerar informações mais completas sobre os programas escritos em Java.

A solução adotada, e apresentada no Capítulo 4, gera classes que contêm metainformações de programas Java como no mecanismo de reflexão. Porém, estas classes são geradas a partir de um arquivo XML que contém informações sobre o código fonte. Estas informações são obtidas por meio da análise sintática e semântica do programa Java.

A desvantagem da solução adotada é que as informações disponíveis para a identificação de oportunidades de refatoração podem não estar em sincronismo com o programa em execução, pois foram obtidas por uma análise anterior do código fonte. Se o mecanismo de reflexão da linguagem Java fosse utilizado para fornecer as informações sobre os programas este problema não ocorreria.

2.3 Programação com regras em linguagem Java

2.3.1 Introdução

Sistemas baseados em regras, ou sistemas especialistas, são programas de computador criados para resolver problemas de interesse específico de um domínio de aplicação ou de uma área de negócios. Para isso, eles emulam o raciocínio de um especialista humano, aplicando seus conhecimentos e inferências.

Para emular o raciocínio e heurísticas humanas, em geral, um sistema especialista funciona da seguinte maneira [Piz04].

- Por algum motivo uma decisão deve ser tomada. Diagnosticar uma doença, por exemplo.
- O sistema tomará suas decisões baseadas em fatos (acontecimentos) lógicos, neste caso, os sintomas da doença.
- Para poder fundamentar sua decisão, o sistema especialista precisa de todo o conhecimento que garanta seu diagnóstico ou dos passos a partir dos quais as informações necessárias são obtidas.

- Durante o processo de raciocínio, vai verificando qual a importância dos fatos que encontra comparando-os com as informações já contidas no seu conhecimento acumulado (base de conhecimento) sobre esses fatos e hipóteses. Neste processo, vai formulando novas hipóteses e verificando novos fatos; e esses novos fatos vão influenciar no processo de raciocínio.
- Este raciocínio é sempre baseado no conhecimento prévio acumulado.
- Um sistema especialista com esse processo de raciocínio pode não chegar a uma decisão se os fatos de que dispõe para aplicar o seu conhecimento prévio não forem suficientes. Pode, por este motivo, inclusive chegar a uma conclusão errada; mas este erro é justificado em função dos fatos que encontrou e do seu conhecimento acumulado previamente.
- Um sistema especialista deve, além de inferir conclusões, ter capacidade de aprender novos conhecimentos e, desse modo, melhorar o seu desempenho de raciocínio e a qualidade de suas decisões.

Normalmente, utilizamos um sistema especialista para preservar o conhecimento sobre determinado assunto ou, ainda, quando o desenvolvimento algorítmico do problema é muito complexo e requer o uso de heurísticas para aumentar sua eficiência. Outro motivo é que decisões humanas podem ser influenciadas pelo ambiente externo, enquanto que as decisões do programa somente serão influenciadas pelas regras programadas.

Existem iniciativas para incluir facilidades para a criação de sistemas especialistas na linguagem Java. Estas facilidades são descritas por uma Especificação de Requisição Java (JSR – *Java Specification Request*). A JSR é uma proposta para especificação a ser incluída pela linguagem Java. A JSR-94 especifica o suporte para programação com máquinas de regras.

A seguir é descrito em linhas gerais o funcionamento de máquinas de regras que seguem esta especificação. No Capítulo 4, a ferramenta Drools [Dro05] que implementa esta especificação é apresentada.

2.3.2 Máquina de Regras

Uma máquina de regras, também conhecida como máquina de decisões ou motor de inferências de regras, é um interpretador sofisticado de instruções *if/then*.

As instruções que a máquina interpreta são chamadas de regras. A parte *if* da instrução define cláusulas como *se cargo = gerente*. A parte *then* contém ações a serem executadas, tais como *aprovarDisconto(15%)*. As ações são disparadas caso a cláusula na parte *if* seja testada como verdadeira. [Jcp04].

As entradas de uma máquina de regras podem ser um conjuto de regras para execução e alguns objetos, também conhecidos como *fatos lógicos*. As saídas, também conhecidas como *conclusões* ou *inferências*, são determinadas pelas entradas e podem incluir: os objetos de entrada originais com algumas possíveis modificações; novos objetos; ou o disparo de ações como inclusões de dados em banco de dados ou envio de mensagens. Os fatos lógicos representam o estado do domínio da aplicação a ser analisado.

Uma máquina de regras típica possui as seguintes características [Jcp04]:

- Promover programação declarativa por meio de uma camada que externaliza a lógica de negócios da aplicação.
- Reconhecer um formato de arquivos ou integrar-se a ferramentas externas à aplicação de edição de regras para execução.
- Agir sobre os objetos de entrada para produzir os objetos de saída.
- Executar ações diretamente, afetando o domínio da aplicação, os objetos de entrada, o ciclo de execução ou a própria máquina de inferências.
- Criar objetos de saída e delegar sua interpretação e execução para o programa que invocou a máquina de regras [Jcp04].

Uma máquina de regras pode funcionar de duas formas: cadeia de retrocesso (*backward chaining*) e cadeia de progressiva (*forward chaining*). Cadeia de retrocesso é utilizada em sistemas baseados em consultas. Neste tipo de cadeia, as máquinas de decisão

consultam o domínio de fatos lógicos para saber se alguma coisa retorna verdadeiro [Drfaq06]. A linguagem PROLOG [Clo94] é originária de um sistema baseado em cadeia de retrocesso.

Já as cadeias progressivas são baseadas em eventos e as conseqüências são disparadas de acordo com o estado do domínio dos fatos lógicos. Este tipo de sistema é declarativo pois seus eventos informam o que deve ser testado no domínio de fatos lógicos [Drfaq06].

Sistemas declarativos conseguem testar domínios de fatos lógicos complexos por meio de linguagens não determinísiticas nas quais um simples evento (eventos definem *o que* deve ser testado) pode ser utilizado para resolver múltiplos problemas, em diferentes componentes (componentes estão fortemente amarrados com o conceito de como fazer) [Drfaq06].

Observe-se no exemplo abaixo o comportamento de um sistema declarativo:

1. Se estiver chovendo;
2. Então você precisa de um guarda-chuva;
3. Se você precisa de um guarda-chuva;
4. Então pegue-o no closet

De acordo com as regras o conhecimento do estado de chuva pode ser trabalhado de duas formas:

1. Você está pronto e já tem um guarda-chuva na mão;
2. Você não tem o guarda-chuva, então vá ao closet e pegue um.

Isto permite que se trabalhe o domínio de fatos lógicos em mais de uma forma e que se responda ao evento no melhor estilo possível, dependendo do conhecimento acumulado [DrD05].

2.3.3 Sistemas baseados em regras e a identificação de oportunidades de refatoração

Sistemas baseados em regras são soluções que visam reduzir o custo associado à incorporação da lógica de negócios nas aplicações, bem como reduzir o custo para a construção de ferramentas e de serviços para a manipulação da lógica de negócios [Jcp04]. Neste cenário, as regras informam à aplicação se ela pode ou não continuar em seu fluxo normal em função das condições testadas e das ações tomadas.

Esta pesquisa utiliza uma máquina de decisões feita em Java, e em conformidade com a especificação JSR-94, para ajudar desenvolvedores a descobrir pontos que possam representar oportunidades para a aplicação de refatorações. A decisão sobre a existência da oportunidade de refatoração é guiada pelas regras que representam o conhecimento humano sobre o assunto tratado, no caso, desenvolvimento de sistemas orientados a objetos.

2.4 Trabalhos relacionados

Nesta seção são discutidos dois trabalhos relacionados com a identificação de automática de oportunidades implementada no SOCRATES. O primeiro é a ferramenta SOUL [Tou03] que identifica oportunidades de refatoração em programas escritos em SmallTalk [Gol89] a partir de consultas realizadas por meio de uma linguagem semalhante à linguagem PROLOG [Clo94]. O segundo trabalho utiliza uma abordagem baseada em métricas para a identificação de maus cheiros e oportunidades de refatoração.

2.4.1 SOUL

SOUL [Tou03] foi desenvolvida para identificar oportunidades de refatoração em sistemas desenvolvidos em SmallTalk. Esta ferramenta é baseada na técnica de lógica de meta-programação (LMP – *Logic Meta Programming*). LMP é a simbiose entre linguagens de programação Orientadas a Objetos (OO) (e.g., SmallTalk, Java) e uma meta linguagem declarativa.

Usando LMP, é possível entender e manipular programas OO de uma forma intuitiva. Esta técnica é empregada para verificar e reforçar as convenções e melhores práticas de projeto, detectar os padrões de projeto utilizados no código, especificar e entender a evolução do projeto e verificar a conformidade da implementação com a arquitetura proposta [Tou03].

A técnica de LMP é independente da linguagem base; porém, a descrição apresentada a seguir é baseada na linguagem SmallTalk.

SOUL e PROLOG

Essencialmente, SOUL é uma meta linguagem declarativa para realizar consultas sobre código escrito em SmallTalk. Trata-se de uma variação da linguagem PROLOG, com algumas pequenas diferenças sintáticas descritas abaixo.

Sintaxe

```
      % → comentários
      vírgula → conjunção lógica
      ?a → variáveis lógicas são sempre precedidas de interrogação (diferente de
PROLOG)
```

Exemplos

```
      %fatos lógicos
      subclass(TermVisitor,SimpleVisitor).
      subclass(SimpleVisitor,NamedVariableVisitor).

      %Regras lógicas
      hierarchy(?P,?C) :- subclass(?P,?C).
      hierarchy(?P,?C) :- subclass(?P,?D), hierarchy(?D,?C)
```

Estas regras lógicas definem que `?P` é ancestral de `?C`, se `?C` é subclasse de `?P`; ou que existe uma classe intermediária `?D` que é subclasse de `?P` e ancestral e `?C`. Dessa maneira, consultas lógicas podem ser realizadas a partir das cláusulas acima.

```
class TermVisitor{
      public abstrac void objectVisit(Object anObjetc);
}

class FixVisitor extends TermVisitor{
      public void objectVisit(Object anObjetc){
            //não usa o parâmetro
      }

}

class SimpleVisitor extends TermVisitor{
      public void objectVisit(Object anObjetc){
            //não usa o parâmetro
      }

}
```

Figura 2.3 – Hierarquia de classes.

Considere-se o trecho de código (escrito em Java para facilitar o entendimento) que descreve uma hierarquia de classes apresentado na Figura 2.3. Por exemplo: `hierarchy(TermVisitor,?C)` determina quando uma subclasse de `TermVisitor` existe e guarda o resultado na variável `?C` (neste caso podem ser tanto `?C= SimpleVisitor`, quanto `?C=FixVisitor`) a consulta `hierarchy(TermVisitor, FixVisitor)` verifica se `FixVisitor` é possivelmente uma subclasse de `TermVisitor`, ainda que indiretamente, e retorna verdadeiro.

Fatos lógicos virtuais

A diferença essencial entre SOUL e outras LMPs é o uso de fatos lógicos virtuais. Todas as entidades do código fonte OO (e.g., classes, métodos, variáveis, relações de herança) podem ser diretamente acessadas dentro do ambiente SOUL por meio de uma meta-interface de mapeamentos representativos de predicados. A vantagem dessa abordagem é que se tem apenas um repositório de fatos lógicos e não dois, caso os fatos do código fossem extraídos do código em tempo de compilação.

Dessa forma, garante-se que sempre a versão mais recente (o próprio código) será analisada, evitando-se problemas de consistência. A Tabela 2.1 apresenta exemplos de mapas de representação de predicados.

Tabela 2.1 – Mapas de representação de predicados utilizados pela ferramenta SOUL [Tou03].

Mapa de representação de predicados	Descrição
`Class(?C)`	C deve ser uma classe
`Hierarchy(?P,?C)`	C é subclasse de P, possivelmente indireta
`ClassImplements(?C,?M)`	C implementa o método M
`InstanceVariable(?C,?V)`	V deve ser uma variável de instancia de C

Estes predicados são usados como predicados lógicos comuns, verificando e recuperando informação. Os autores não mencionam, mas os fatos lógicos virtuais devem utilizar o mecanismo de reflexão nativo da linguagem em que a SOUL será aplicada, a saber, SmallTalk.

Detectando cheiros

É apresentado abaixo um exemplo em que a técnica LMP, implementada em SOUL, pode ser usada na identificação de cheiros que indiquem oportunidades de refatoração.

Parâmetro obsoleto é um parâmetro definido por um método em sua assinatura e que nunca é usado. Este é um problema de difícil detecção pois é muito trabalhoso conhecer o interior de todas as classes que sobrescrevem o método de parâmetro obsoleto, a fim de certificarmos que tal parâmetro realmente nunca é usado.

Para detectar quando um parâmetro formal não é usado, deve-se verificar se o próprio método, e nem um de seus sobrescritores, o está usando. Podem ser usadas as duas regras lógicas subseqüentes para implementar tal algoritmo:

```
ObsoleteParameter(?class,?selector,?parameter) :-
[1]classImplements(?class,?selector),
[2]parameterOf(?class,?selector,?parameter),
[3]forall(subclassImplements(?class,?selector,?subclass)),
[4]not(selectorUsesParameter(?subclass,?selector,?parameter))
```

Inicialmente, os métodos implementados por uma dada classe são recuperados [linha 1] (o predicado `classImplements` é parte do mapeamento de representação de predicados), depois todos os parâmetros são recuperados [linha 2] (predicado `parameterOf`). Para cada subclasse que implementa o dado método da classe [linha 4] é feita a a verificação se o parâmetro está sendo usado.

O predicado `selectorUsesParameter` é implementado como se segue:

```
selectorUsesParameter(?class,?selector,?parameter) :-
[1]classImplementsMethodName(?class,?selector,?method)
[2]parseTreeUsesVariable(?method,?parameter)
```

Está sendo usada uma variante do predicado `classImplements`, o predicado `classImplementsMethodName` que retorna a árvore sintática (`parseTree`) do método identificado pela classe e o selector [linha 1], o predicado `parseTreeUsesVariable` então, é utilizado para percorrer a `parseTree` e procurar por usos do parâmetro especificado [linha 2]. Este predicado é definido em termos do predicado `traverseMethodParseTree` que implementa um algoritmo de busca em árvore

Propondo refatorações

Uma vez que os maus cheiros foram identificados, devem ser indicados as refatorações que podem solucioná-los. Para indicar a refatoração adequada, é utilizado o predicado `proposeRefactoring`. O predicado possui a seguinte forma:

```
proposeRefactoring(?entity,?refactoring,?arguments)
```

O primeiro argumento identifica a entidade que será analisada para se descobrir quais são as oportunidades de refatoração. Pode ser qualquer artefato de código fonte, mas, na referência utilizada, apenas classes, métodos ou variáveis de instância são permitidas haja vista que estas são as entidades para as quais existem refatorações definidas.

O segundo argumento identifica a refatoração em particular que deve ser aplicada.

Seu valor poderia ser `addClass`, `pullUpMethod`, `abstractVariable` ou qualquer refatoração definida. O último argumento do predicado identifica a lista de argumentos que deveriam ser passados para serem refatorados, estes argumentos poderiam ser qualquer artefato de código fonte.

O mau cheio *parâmetro obsoleto* pode ser tratado aplicando-se a refatoração `removeParameter` conforme indicado abaixo.

```
      proposeRefactoring(?class,removeParameter,<?class, ?selector, ?parameter>)
:- obsoleteParameter(?class,?selector, ?parameter)
```

Esta refatoração visa garantir que o parâmetro obsoleto seja retirado do método onde foi definido, de todos os métodos sobrescritores e de todos os métodos chamadores.

2.4.2 Identificação de oportunidades de refatoração por meio de métricas

Identificar o trecho de código – isto é, identificar oportunidades de refatoração –que precisa ser modificado é a base da atividade de refatoração. Carneiro e Mendonça [Car03] propõem o uso de métricas para auxiliar a identificação de oportunidades de refatoração.

O objetivo do seu trabalho é viabilizar uma forma quantitativa de analisar os resultados obtidos da aplicação de métricas visando direcionar a aplicação de refatorações. Este tipo de análise é importante, pois as métricas permitem uma análise menos subjetiva e menos sujeita a heurísticas pessoais.

Os autores apresentam duas abordagens que visam à *definição* e à *identificação* de métricas úteis para identificação de oportunidades de refatoração. A abordagem chamada *top-down* é baseada no método *Meta Pergunta Métrica* (MPM ou GQM – *Goal Question Metric*) [Pig97]. Já a abordagem *bottom-up* baseia-se na análise empírica do relacionamento entre métricas, refatorações e maus cheiros encontrados em estudos de caso.

Abordagem *top-down* (baseada na técnica Meta Pergunta Métrica)

Esta abordagem utiliza a técnica Meta Pergunta Métrica para identificar métricas que detectem oportunidades de refatoração. Os maus cheiros identificados por Fowler e colegas [Fow99] são o subsídio para identificar métricas conhecidas, ou definir novas métricas, que sejam úteis para a identificação de oportunidades de refatoração.

O primeiro passo da abordagem é saber quais são as oportunidades de refatoração que se deseja identificar. Por exemplo, considere-se que seja desejado identificar/definir métricas para caracterizar o mau cheiro *código duplicado*. As ocorrências mais comuns desse mau cheiro e as ações de correção associadas a ele são apresentadas na Tabela 2.2:

Tabela 2.2 – Possíveis ocorrências do mau cheiro código duplicado e sugestões de refatoração [Car03].

Possíveis ocorrências de código duplicado	**Sugestão de correção por meio de refatoração**
A mesma expressão existe em dois métodos da mesma classe.	*Migração do Trecho de Código para um novo Método*.
A mesma expressão existe em duas subclasses da mesma classe.	*Migração do Trecho de Código para um novo Método* e *Migração de Variável para uma Superclasse*, respectivamente.
Duas ou mais ocorrências de código similar.	*Migração do Trecho de Código para um novo Método* e *Migração do Trecho de Código para um novo Método Modelo*, respectivamente.
Os métodos possuem o mesmo objetivo com algoritmos diferentes.	Escolher o melhor dos dois algoritmos e depois usar a refatoração *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.

Em seguida deve-se estabelecer uma meta: *Analisar o código fonte com o objetivo de melhorá-lo (refatorá-lo) com respeito a código duplicado do ponto de vista do programador e da equipe de manutenção*.

O próximo passo é estabelecer as perguntas que auxiliem a identificação/definição das métricas a fim de que a oportunidade de refatoração possa ser sistematicamente

encontrada e corrigida. A Tabela 2.3 mostra as perguntas e métricas para o mau cheiro *código duplicado*.

Tabela 2.3 – Perguntas e métricas derivadas para o mau cheiro código duplicado [Car03].

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

Carneiro e Mendonça aplicaram a abordagem *top-down* em 16 maus cheiros contidos no catálogo de Fowler e colegas e obtiveram 36 conjuntos de métricas (M1.1 a M1.4 na Tabela 2.3 representam métricas definidas/identificadas). Sete desses conjuntos são compostos de métricas simples e já disponíveis; dois conjuntos contêm métricas deriváveis a partir de métricas simples e disponíveis. Os outros 27 conjuntos são de métricas não disponíveis, sendo que seis desses conjuntos contêm métricas fortemente dependentes da análise cognitiva, como a métrica **M1.3** da Tabela 2.3.

Abordagem *bottom-up* (Análise Empírica)

Esta abordagem visa complementar a abordagem *top-down* acima. A abordagem *bottom-up* é, basicamente, a verificação empírica de quais métricas foram alteradas em função das refatorações aplicadas, assim como das oportunidades de refatoração correspondentes.

Sendo assim, seu principal objetivo é verificar a aplicabilidade da abordagem *top-down*, a saber: verificar se o que foi previsto aconteceu, verificar se as métricas variaram de forma inesperada e realimentar as heurísticas aplicadas.

Resultados do estudo de caso

Os autores realizaram um estudo de caso com uma aplicação fornecida pelo próprio Martin Fowler [Fow05]. Os resultados obtidos mostram que existe uma relação entre métricas e refatorações, isto é, ocorre variação nos valores das métricas antes e depois da aplicação de uma refatoração. Porém, não foi possível estabelecer a mesma relação entre métricas e maus cheiros no estudo de caso. Os autores afirmam que o estudo de caso realizado é ainda preliminar e não é possível tirar conclusões sem a condução de estudos adicionais.

2.5 Considerações finais

Neste capítulo, foram apresentados os conceitos de refatoração e mau cheiro, essenciais neste trabalho. Refatoração, segundo Fowler e colegas [Fow99], é uma mudança interna na estrutura do software a fim de tornar mais fácil o seu entendimento e a sua modificação mais barata, sem, contudo, modificar o seu comportamento observável.

Maus cheiros, por sua vez, representam problemas existentes no código que dificultam a sua manutenção e evolução e que são solucionados pela aplicação de uma ou mais refatorações. Os maus cheiros constituem oportunidades para a aplicação de refatorações e sua identificação automática é o objeto desse trabalho.

Para atingir este objetivo, foram apresentados os conceitos e meta-programação, *reflexão* e *programação baseado em regras em Java*. O SOCRATES (Sistema Orientado a objetos para CaRActerização de refaToraçõES) utiliza estes conceitos para implementar a identificação de oportunidades de refatoração. Foram ainda apresentados dois trabalhos, com abordagens distintas, para identificação de oportunidades de refatoração.

O primeiro é baseado no conceito de Lógica de Meta-programação (LMP – *Logic Meta Programming*) que permite que linguagens de programação do estilo de programação em lógica realizem consultas ao código de programa escritos em linguagens orientadas a

objeto. Sua principal vantagem é permitir expressar de maneira elegante e concisa algoritmos para determinar a presença de determinados maus cheiros. No entanto, para realizar esta tarefa, requer em alguns casos (como o mau cheiro *parâmetro obsoleto*), consultar a árvore sintática dos programas, o que não é realizado eficientemente utilizando programação em lógica. Além disso, a abordagem puramente algorítmica não é suficiente para determinar a presença determinados mau cheiros (e.g., *código duplicado*).

Para estes casos, a segunda abordagem apresentada pode ser uma solução visto que ela é baseada em métricas para ajudar o programador a eliminar parte do código a ser analisado. Dessa maneira, a atenção do programador pode ser direcionada para os trechos de código mais sujeitos a presença de determinados maus cheiros. Entretanto, esta abordagem depende da realização de vários estudos de casos que possibilitem a criação de heurísticas que direcionem o programador. Infelizmente, a realização desses estudos está em estágio preliminar.

A abordagem implementada no sistema SOCRATES, apresentada por meio de um exemplo simplificado no próximo capítulo, é semelhante à abordagem que utiliza lógica de meta-programação. No entanto, SOCRATES procura utilizar uma abordagem leve (*lightweight*) para atingir o mesmo objetivo. O sistema é baseado em ferramentas livres que realizam tarefas de infra-estrutura (coleta de informações sobre o código fonte e tratamento de regras) e concentra a programação adicional em determinados pontos. Dessa maneira, evita-se a programação de um ambiente de programação em lógica (como no caso da ferramenta SOUL) e os algoritmos que precisam ser implementados podem ser realizados da maneira mais eficiente possível.

Capítulo 3

SOCRATES - Sistema Orientado para CaRActerizacao de refaToraçõES

SOCRATES (Sistema Orientado para CaRActerizacao de refaToraçõES) é um protótipo para identificação de possíveis refatorações em programas escritos em linguagem Java. Para atingir seu objetivo, ele recebe como entrada um programa Java que é analisado e mapeado para um arquivo em formato XML. Este arquivo contém informações sobre o código do programa tais como classes e métodos definidos, troca de mensagens entre objetos e definições e usos de variáveis.

A partir dessas informações e da utilização de regras pré-definidas, SOCRATES analisa o programa para identificar pontos candidatos a uma possível refatoração. Estas regras são disparadas por um motor de inferências capaz de guardar estados e informações auxiliares entre um disparo e outro, além de mudar o fluxo de aplicação destas regras dependendo dos resultados obtidos. Desta forma, SOCRATES utiliza recursos de inteligência artificial para realizar a análise do código.

3.1 Motivação

SOCRATES é um sistema que promove análise automatizada de programas de computador. Seu objetivo é emular um especialista que necessita analisar longos trechos de código fonte para decidir que determinado ponto do programa precisa ser reestruturado. A vantagem de SOCRATES é que, diferentemente dos especialistas humanos, não se cansa e não sofre pressões que podem desviar os resultados de sua análise.

A tarefa de identificar o ponto do programa candidato a ser refatorado não é trivial. Considere-se a situação de mau cheiro caracterizada por um *parâmetro obsoleto* em um

método, isto é, um parâmetro que não é mais utilizado. Este mau cheiro é eliminado pela aplicação da refatoração *remove parameter* [Fow99]. Mesmo a identificação de um mau cheiro trivial como um parâmetro obsoleto pode demandar uma investigação detalhada do código.

Isto porque a implementação de um método pode estar espalhada em várias subclasses que o sobrescrevem. Assim, supondo que a classe implemente *m* métodos e que *n* subclasses desta classe sobrescrevam este método, o desenvolvedor precisa inspecionar *m* x *n* métodos para detectar um parâmetro obsoleto. Além disso, essa tarefa deve ser realizada para todo parâmetro formal que o método define. Portanto, detectar a ocorrência de um parâmetro obsoleto é um processo custoso e demorado [Tou03].

O objetivo de SOCRATES é realizar de maneira automática a inspeção do código para identificar pontos do programa que são candidatos a refatoração. Este objetivo é realizado por meio de análise do código fonte e pela aplicação de regras. Dessa maneira, é possível reduzir erros de análise comuns quando esta tarefa é realizada por humanos. Neste capítulo, a estrutura básica do SOCRATES é descrita através da apresentação de seus componentes básicos e de um exemplo simplificado de utilização.

3.2 A anatomia do SOCRATES

O SOCRATES é um programa de computador que tem uma estrutura interna com responsabilidades bem definidas. Esta estrutura é mapeada em quatro componentes de negócios e três de infra-estrutura conforme descrito na Figura 3.1. Os componentes possuem relações de dependências sem ciclos e que definem o fluxo de informações do SOCRATES. Essa seqüência, apresentada na Figura 3.2, estende-se desde o momento em que o código fonte é lido até o momento em que se encontram os pontos candidatos a refatoração. Estes componentes foram construídos de forma que os efeitos dessa relação de dependência sejam minimizados e os componentes possam ser substituídos ou customizados sem que os impactos desta operação sejam desconhecidos. A seguir são descritos os componentes do SOCRATES.

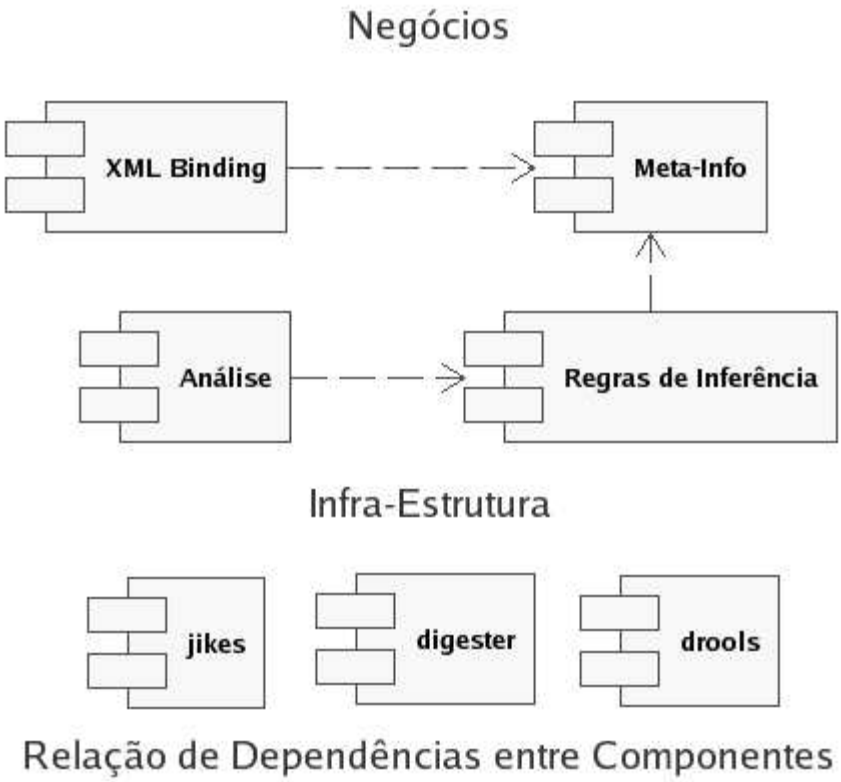

Figura 3.1– Estrutura de componentes do SOCRATES.

Extração da Meta-Informação - Jikes

A ferramenta de infra-estrutura Jikes [Jik05] é responsável por extrair do programa Java (que está sendo analisado quanto à possível aplicação de refatorações) informações a respeito da sua estrutura. As informações que são coletadas a respeito do programa são:

- Informações Estruturais
  - Nome de pacotes, classes, atributos, métodos, parâmetros e variáveis locais.
- Informações Comportamentais
  - Nome de classes importadas, declaração de objetos e envio de mensagens.
- Localização de Arquivos
  - Localização do arquivo original Java em disco de acordo com as convenções do sistema operacional utilizado.

Estas informações estarão disponíveis em formato XML e descrevem o mesmo programa Java em uma representação alternativa, porém, mais facilmente manipulável por outras ferramentas. Os rótulos do arquivo XML gerado pela ferramenta Jikes segue o formato JavaML criado por Badros [Bad00]. A Figura 3.4 apresenta o resultado do processamento do trecho de programa contido na Figura 3.3 pelo Jikes. No próximo capítulo o funcionamento da ferramenta Jikes é detalhado.

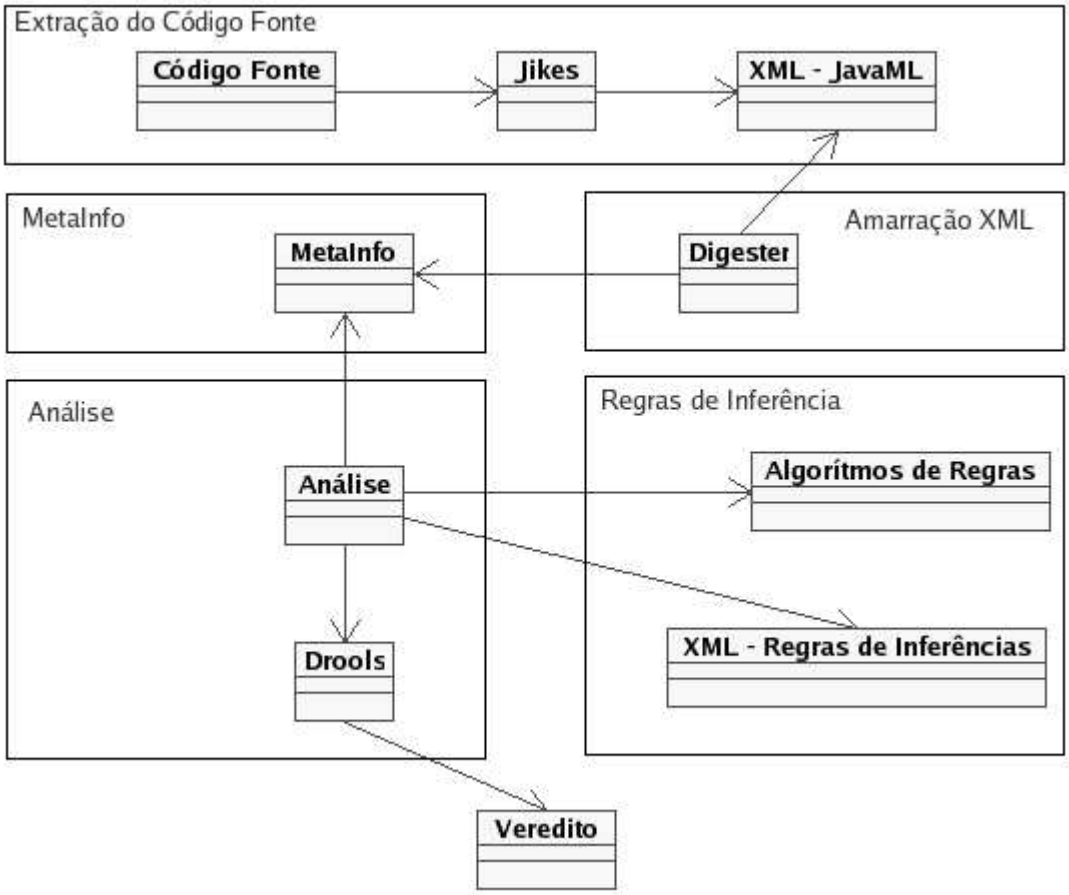

Figura 3.2 – Fluxo de informações dos componentes do SOCRATES.

```
public class Exemplo{

	String variavel = "";

	public void metodo(String parametro){
		variavel = "Parametro e obsoleto !";
	}
}
```

Figura 3.3 – Trecho de programa Java.

```
1: <?xml version="1.0" encoding="UTF-8"?>
2: <!DOCTYPE java-source-program SYSTEM "java-ml.dtd">

3: <java-source-program>
4: <java-class-file name="C:/jikes/socrates/Exemplo.java">
5: <class name="Exemplo" visibility="public" line="1" col="0" end-line="9" end-
col="0">
6:  <superclass name="Object"/>
7:  <field name="variavel" line="3" col="8" end-line="3" end-col="28"><type
name="String"/><literal-string value=""/></field>
8:  <method name="metodo" visibility="public" id="Exemplo:mth-13" line="5"
col="8" end-line="7" end-col="8">
9:    <type name="void" primitive="true"/>
10:    <formal-arguments>
11:      <formal-argument name="parametro" id="Exemplo:frm-11"><type
name="String"/></formal-argument>
12:    </formal-arguments>
13:    <block line="5" col="44" end-line="7" end-col="8">
14:      <assignment-expr op="="><lvalue><var-set
name="variavel"/></lvalue><literal-string value="Parametro e obsoleto
!"/></assignment-expr>
15:    </block>
16: </method>
17:</class>
18:</java-class-file>
19:</java-source-program>
```

Figura 3.4 – Arquivo JavaML gerado pela ferramenta Jikes

Observando a Figura 3.4, nota-se que as informações resultantes da análise sintática e semântica do código estão contidas no arquivo XML gerado por Jikes. Por exemplo, na linha 4, o rótulo `<java-class-file ...>` indica a localização em disco do arquivo original Java. Os atributos do rótulo `<class ...>` (linha 5) indicam o nome da classe definida, no caso "`Exemplo`", sua visibilidade `(public)`, bem como a sua localização no código fonte (atributos `line`, `col`, `end-line`, `end-col`). Outras informações relativas ao método declarado (linhas 8 a 16), ao uso e definição de variáveis (linha 14) estão igualmente definidas utilizando os rótulos do padrão JavaML.

Amarração XML – XML Binding

O componente *Amarração XML* (XML *Binding*) é o responsável por tornar disponível para os outros componentes do SOCRATES as informações estruturais disponíveis no arquivo JavaML gerado por Jikes. Isto é realizado no SOCRATES *amarrando-se* o conteúdo do arquivo JavaML resultado da compilação do programa pelo Jikes a *objetos* Java acessíveis pelos demais componentes do SOCRATES.

Para a realização dessa amarração, SOCRATES utiliza o componente de infra-estrutura Digester [Dig05]. O componente Digester lê o arquivo JavaML e cria uma classe Java que contém as informações estruturais necessárias para a análise do código. A classe gerada pelo Digester é instanciada em um objeto capaz de ser acessado pelos demais componentes SOCRATES.

O Digester [Dig05] é um *framework* criado pelo projeto *Jakarta Commons* da *Apache Software Foundation* (ASF) com o objetivo de configurar arquivos XML de forma que possam ser mapeados em objetos Java. A Figura 3.5 contém a classe Java (esta figura e as que se seguem foram obtidas a partir da interface do ambiente de programação Eclipse [Ecl05]) mapeada pelo componente Digester a partir do arquivo JavaML contido na Figura 3.3. Uma descrição mais detalhada do componente Digester é apresentada no próximo capítulo.

A classe Java mapeada é a classe `Metainfo.java` descrita a seguir. Esta classe, indicada na Figura 3.5 pela letra C maiúscula e seu nome (`Metainfo`), é composta por uma lista de objetos da classe `File` (arquivo) que representam os arquivos físicos em disco.

Figura 3.5 – Classe Java mapeada pelo componente Digester a partir do arquivo JavaML.

A classe `File`, por sua vez, representada na Figura 3.6 a partir da letra C, possui um nome, que representa sua localização física em disco, uma lista de objetos `Import` – informações sobre as classes que devam ser utilizadas internamente; um objeto do tipo `Package` – informações sobre a localização lógica; e um objeto `Class` – a classe que o arquivo em disco contém.

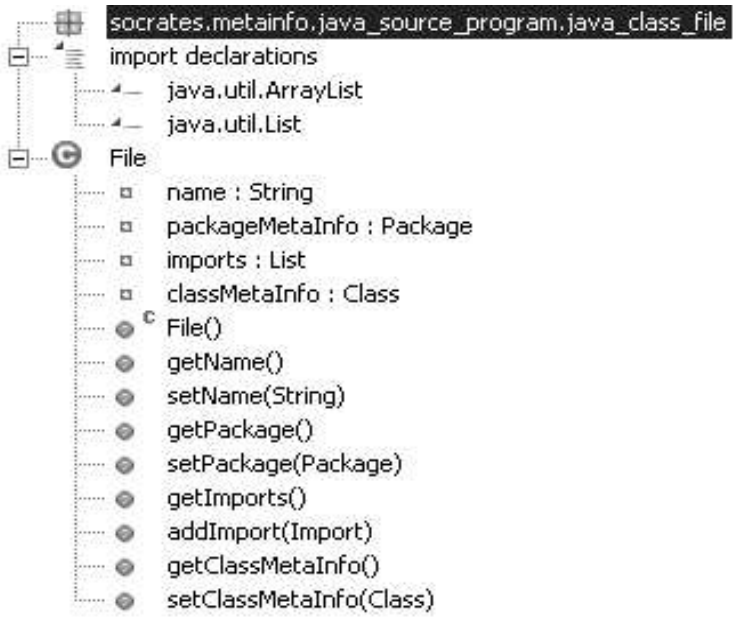

Figura 3.6 – Classe que representa o arquivo Java em disco a ser analisado.

Por fim, é necessário examinar a classe `Class` para entendermos a forma como os atributos do arquivo JavaML são mapeados em objetos Java. Esta classe está representada na Figura 3.7; é possível perceber que ela armazena as informações sobre todos os elementos utilizados internamente na classe de forma lógica, ou seja, informações sobre métodos, parâmetros, variáveis, chamadas de outros objetos e informações de hierarquia de heranças.

Figura 3.7 – Classe que representa as meta-informações da classe Java a ser analisada.

Meta-info

O componente *Meta-info* é a instanciação em objetos da classe Java mapeada pelo componente *Amarração XML*. Nos objetos, as informações estruturais relativas ao programa Java analisado pelo Jikes e contidas no arquivo JavaML estão disponíveis na forma de atributos. A Figura 3.8 apresenta a instanciação dos objetos que compõe o componente *Meta-info* para o programa da Figura 3.3.

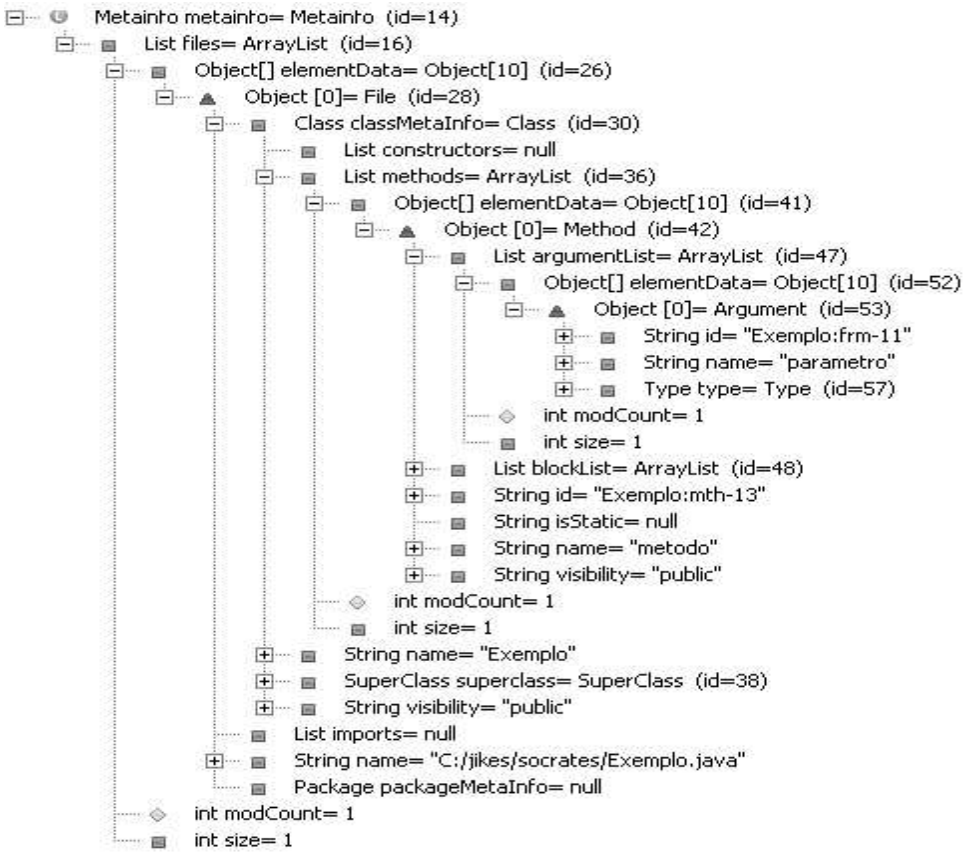

Figura 3.8 – Instanciação da classe Java criada pelo componente Digester para utilização pelo SOCRATES.

Assim, o componente *Meta-Info* contém os itens de análise do programa e suas representações em objetos internos ao SOCRATES. Estes objetos internos serão utilizados pelos outros componentes para identificar as oportunidades de refatoração. Por exemplo, no processo de análise precisamos investigar como um parâmetro e um método se relacionam para decidir se o parâmetro é ou não obsoleto. Assim, método e parâmetro são para o SOCRATES objetos Java que contêm informações de uso em sua forma original no código fonte.

Análise

O componente de análise é o responsável por comandar a execução dos algoritmos que identificam oportunidades de refatoração. Os algoritmos são aplicados aos objetos de

`Metainfo` recém-extraídos. A ferramenta Drools [Dro05] é a infra-estrutura que permite a definição e a operação de regras que invocam os algoritmos de identificação de refatorações.

Drools é uma máquina de regras baseada em linguagens declarativas. Internamente, esta máquina processa as regras através de um grafo em que cada vértice representa uma regra e cada aresta um relacionamento entre elas. Nesta representação, uma regra é um conjunto que contém uma condição e uma conseqüência. Quando a condição for satisfeita, a conseqüência, que contém uma ação a ser realizada, será disparada. Após sua execução, de acordo com o grafo formado, a próxima regra será chamada.

Para construir o grafo, o Drools interpreta os arquivos XML que representam as regras. Algumas instruções especiais dentro do arquivo XML estabelecem os relacionamentos entre as elas. Estes arquivos formam os arquivos *XML de Regras* e todo este processo será detalhado no Capítulo 4. A Figura 3.9 contém o arquivo XML de regras definido para identificar o mau cheiro parâmetro obsoleto.

```
1: <rule name="parametro obsoleto">
2:    <parameter identifier="metainfo">
      3:<java:class>socrates.metainfo.java_source_program.Metainfo</java:class>
4:    </parameter>
5:    <java:condition>metainfo != null</java:condition>
6:    <java:consequence>
7:        socrates.rules.obsoleteparameter.ObsoleteParameter ob = new
socrates.rules.obsoleteparameter.ObsoleteParameter(metainfo);

8:         java.util.List obsoleteParamenters = ob.getObsoleteParameters();

9:           if(!obsoleteParamenters.isEmpty()){
10:              System.out.println("Os seguintes parametros sao obsoletos: ");
11:              for (java.util.Iterator iObsolete =
obsoleteParamenters.iterator(); iObsolete.hasNext();) {

12:socrates.metainfo.java_source_program.java_class_file.class_decl.method.Argume
nt argument =
(socrates.metainfo.java_source_program.java_class_file.class_decl.method.Argument
) iObsolete.next();
13:System.out.println("* "+argument.getName());
14:            }
15:          }else{
16:                System.out.println("Não existem parametros obsoletos");
17:          }
18:    </java:consequence>
19: </rule>
```

Figura 3.9 – Arquivos XML de Regras para identificar o mau cheiro parâmetro obsoleto.

A regra da Figura 3.9 é dividida em 3 seções. A primeira seção no rótulo *parameter* (linhas 2 a 4), define os parâmetros que podem ser passados para a regra. Estes parâmetros funcionam da mesma forma que parâmetros passados na chamada de métodos comuns a linguagem de programação Java; são cópias dos valores de objetos externos que podem ser utilizados nas outras seções da regra. A segunda parte, definida pelo rótulo *condition* (linha 5), informa a condição de ativação da regra, é o momento em que os predicados cruzam com os fatos lógicos, que podem ser objetos do próprio programa a ser analisado. Por fim, o rótulo *consequence* (linhas 6 a 19), informa o que deve acontecer se o teste da condição resultar em verdadeiro.

Na Figura 3.10 pode-se ver a classe `Brain` que realiza a análise. No canto esquerdo, encontra-se a sua estrutura; no canto direito inferior está localizada a divisão em pacotes; e no canto direito superior, estão seus dois métodos mais importantes que fazem o mapeamento do SOCRATES com o componente de infra-estrutura Drools. O método `setup` é responsável por passar o caminho correto dos arquivos XML de regras a serem executadas e o método `activate` dispara a execução destas regras.

Os arquivos XML de Regras possuem uma estrutura definida pela própria ferramenta de infra-estrutura para orquestrar o fluxo de execução dos algoritmos programados em Java que definem a implementação das regras de análise. Estes algoritmos constituem as *Regras de Inferência* discutidas a seguir.

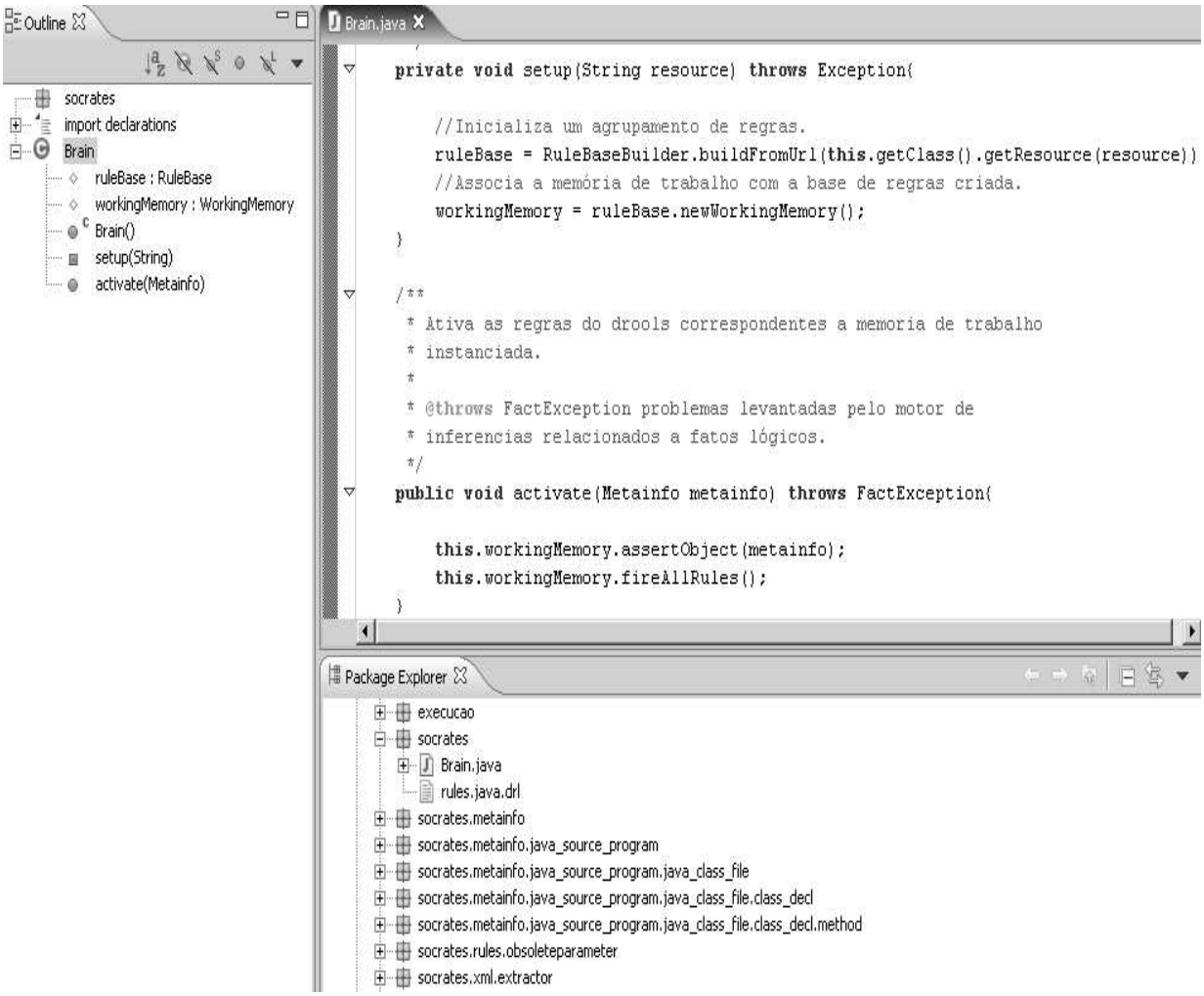

Figura 3.10 – Representação estrutural do componente de análise e sua posição na estrutura de pacotes com todos os componentes do SOCRATES.

Regras de Inferência

As regras de inferência são programas Java que implementam algoritmos para encontrar os mau cheiros. Esses algoritmos são baseados em experiências e heurísticas de um especialista. É a documentação e a garantia de repetição sistemática dos passos que este profissional deve realizar para se chegar a um veredicto. O cruzamento positivo entre as regras de inferência e um objeto da classe `Metainfo.class` indica que um candidato a refatoração foi identificado.

Na Figura 3.9, no rótulo *consequence*, o objeto `metainfo` é passado como parâmetro pelo SOCRATES, que define o conjunto de informações extraídos do código fonte a ser analisado. A condição testa se este objeto é nulo, caso não seja, a conseqüência será ativada. Neste momento o algoritmo que verifica a existência de parâmetro obsoleto (linha 7) (descrito no Capítulo 5) será instanciado e executado. O objeto `metainfo` contém todas as informações relevantes do programa a ser analisado para se chegar a um veredito. Caso algum parâmetro obsoleto seja encontrado o usuário do sistema será informado (linhas 8 a 17).

3.2 Seqüência de atividades do SOCRATES

A partir da combinação da relação de dependências (Figura 3.1), o fluxo de informações (Figura 3.2) e a seqüência de atividades do SOCRATES (Figura 3.11), tem-se que:

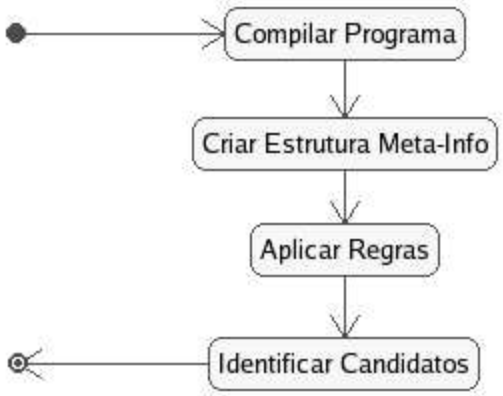

Figura 3.11 – Seqüência de atividades do SOCRATES.

1. O programa fonte Java é analisado pela ferramenta Jikes gerando um arquivo XML com as meta-informações do programa a ser analisado. Este passo é representado pela atividade *Compilar Programa*.

2. O Componente *Amarração XML* recebe como entrada o arquivo JavaML gerado e o amarra a objetos Java capazes de serem manipulados pelo SOCRATES. O componente *Amarração XML* utiliza o Digester para fazer a

amarração com objetos do componente Meta-Info. Este passo é representado pela atividade *Criar Estrutura Meta-Info*.

3. As classes de *Meta-Info* são, então, utilizadas pelos algoritmos de busca de candidatos a refatoração implementados no componente *Regras de Inferênci*a. Os arquivos XML de Regras gerenciam as aplicações dos algoritmos de busca que resultam na identificação positiva ou negativa de oportunidades de refatoração. Este passo utiliza o Drools como infra-estrutura e é representado pela atividade *Aplicar Regras* e *Identificar Candidatos*.

Desta forma, SOCRATES consegue atingir seus objetivos de automatizar algumas atividades humanas que são caras devido à alocação de mão-de-obra para atividades não-produtivas (já que não estará produzindo software) e entediantes, o que leva a erros que podem ser ainda mais caros de serem encontrados do que os erros originais.

O resultado da análise para a identificação de parâmetros obsoletos para o exemplo simplificado contido na Figura 3.1 pelo SOCRATES é indicado na Figura 3.10.

```
Os seguintes parametros sao obsoletos:
* args
```

Figura 3.12 – Resultado da análise pelo SOCRATES para a identificação de parâmetros obsoletos no programa da Figura 3.1.

3.3 Considerações Finais

Este capítulo mostrou a estrutura do SOCRATES e como seus componentes internos trabalham em conjunto com os componentes de infra-estrutura para conseguir identificar oportunidades de refatoração do código fonte analisado.

No próximo capítulo são analisados os componentes de infra-estrutura e de negócio

e como eles conseguem prover os serviços necessários ao SOCRATES para a identificação de oportunidades de refatoração. Além disso, são discutidos aspectos especiais sobre o uso do SOCRATES como a inclusão de novas regras.

Capítulo 4

Detalhes de Implementação

A implementação do SOCRATES (Sistema Orientado para CaRActerizacao de refaToraçõES) é baseada na utilização de componentes de infra-estrutura fornecidos por ferramentas livres, ou seja, ferramentas de domínio público mas sujeitas a licença de uso de seu mantenedor. A disponibilidade dessas ferramentas permitiu que SOCRATES fosse desenvolvido rapidamente e com pouca codificação adicional. Neste capítulo, serão apresentadas as ferramentas utilizadas, bem como os detalhes de implementação que permitem a comunicação entre elas e a detecção do mau cheiro *parâmetro obsoleto*. Adicionalmente, são também discutidos os procedimentos para a inclusão de novas regras para a detecção de maus cheiros adicionais, isto é, para identificação de oportunidades de refatoração. Um exemplo de utilização do SOCRATES em uma de suas classes é incluído.

4.1 Componentes de infra-estrutura

Como apresentado no capítulo anterior, o SOCRATES possui três componentes de infra-estrutura, a saber, J*ikes*, *Digester* e *Drools*. Estes três componentes são implementados por funcionalidades fornecidas pelas ferramentas livres de mesmo nome que são descritas a seguir.

4.1.1 Jikes

Jikes é um compilador Java desenvolvido inicialmente pela IBM (*International Business Machines*). Hoje, porém, é um projeto de software livre, podendo ser obtido no

sítio *SourceForge* [Jik05]. Badros [Bad00] modificou a versão original do compilador para, ao invés de criar *bytecodes*, criar arquivos XML que representam as meta-informações do código compilado.

As meta-informações foram mapeadas para arquivos XML que seguem o formato JavaML [Bad00]. Este formato é uma especialização do padrão XML realizada por meio de um arquivo DTD – *Document Type Definition* [Har01]. O arquivo DTD define a estrutura do formato JavaML, bem como uma lista de elementos (rótulos, atributos) aceitos. Com a descrição contida no arquivo JavaML DTD, ferramentas que processam arquivos XML (como a ferramenta Digester descrita na Seção 4.1.2) podem analisar as meta-informações geradas por Jikes e armazenadas no arquivo JavaML.

Para obter as meta-informações, foi adicionado ao Jikes o método `XMLUnparse` em cada um dos nós da árvore sintática abstrata (AST – *Abstract Sintax Tree*) gerada no processo de compilação. Foram incluídos cerca de 1650 termos de correspondência entre a linguagem Java e os elementos do arquivo DTD JavaML para possibilitar a conversão entre os arquivos.

O conversor funciona utilizando a infra-estrutura do compilador como o motor que percorre o programa e colhe as informações necessárias para o trabalho de conversão. Ele foi largamente testado e não foram encontrados defeitos nas conversões Java para JavaML ou JavaML para Java e nem mesmo diferenças nos programas convertidos e re-convertidos em qualquer uma das representações.

Para entender melhor o funcionamento e as vantagens do Jikes considere o exemplo da Figura 4.1:

```
import java.applet.*;
import java.awt.*;
public class FirstApplet
extends Applet {
      public void paint(Graphics g) {
      g.drawString("FirstApplet", 25, 50);
      }
}
```

Figura 4.1 – Programa exemplo [Bad00].

O compilador é invocado através da linha de comando de forma padrão: `jikes Applet.Java`. Assim, ao invés de um *bytecode* o seguinte arquivo JavaML é gerado como saída:

```
<java-source-program>
      <import-declaration>import java.applet.*;</import-declaration>
      <import-declaration>import java.awt.*;</import-declaration>
      <class-declaration>
          <modifiers>public</modifiers> class
          <class-name>FirstApplet</class-name>
          extends
          <superclass>Applet</superclass> {
          <method-definition>
                <modifiers>public</modifiers>
                <return-type>void</return-type>
                <method-name>paint</method-name>
                (<formal-arguments>
                <type>Graphics</type>
                <name>g</name>
                </formal-arguments>)
                <statements>{
                     g.drawString("FirstApplet", 25, 50);
                } </statements>
           </method-definition>
      }
      </class-declaration>
</java-source-program>
```

Figura 4.2 – Arquivo JavaML gerado pelo processo de compilação do Jikes com o método XMLUnparse.

As meta-informações contidas no arquivo JavaML criado por Jikes dizem respeito à estrutura do código da classe analisada. No exemplo da Figura 4.2, pode-se observar que foram incluídas no arquivo JavaML o nome da classe definida, sua superclasse, seus modificadores, etc. Com relação aos métodos definidos, estão incluídas informações sobre os seus valores de retorno, parâmetros formais e o próprio código dos métodos.

Estas informações descritas em JavaML tipicamente são obtidas pelo *front end* dos compiladores Java. A solução proposta por Badros [Bad00] foi tornar estas informações disponíveis em um formato conciso padrão (JavaML) e que podem ser processadas facilmente por outras ferramentas. Em especial para SOCRATES, informações tais como os parâmetros utilizados pelos métodos de uma classe e as variáveis utilizadas no corpo do método são necessários para identificar o mau cheiro *parâmetro obsoleto*.

A utilização do Jikes foi necessário porque a idéia original de utilizar o mecanismo de reflexão da linguagem Java não foi possível na versão 1.4 da linguagem. Dessa maneira, Jikes modificado por Badros [Bad00] fornece as meta-informações necessárias para a identificação de oportunidades de refatoração.

4.1.2 Digester

Para realizar o mapeamento entre o arquivo JavaML gerado pelo componente Jikes e a classe de meta-informações, SOCRATES utiliza a ferramenta Digester [Dig05] como componente de infra-estrutura. O resultado da utilização do Digester é o mapeamento dos elementos (rótulos, atributos) e valores (valores dos atributos, conteúdo dos rótulos) presentes no arquivo JavaML em elementos (e.g., classes, métodos, atributos) e valores (e.g., valores de atributos) da estrutura de classes internas que representam as meta-informações. Esta operação é chamada de *Amarração XML* (*XML Binding*).

Digester é um componente do subprojeto Jakarta Commons [Dig05] que suporta processamento dirigido a eventos para documentos XML de contexto geral. Eventos são ações sobre qualquer elemento dentro de arquivos XML tais como rótulos, atributos e definições de espaços de nomes. Com Digester é possível fazer a amarração entre um arquivo XML e um objeto Java através do disparo de regras que definem as transformações. Estas regras são disparadas de acordo com o mapeamento feito entre eventos e objetos. Assim, o evento início de rótulo sobre o objeto <*java-source-program*> do arquivo XML da Figura 4.2 dispara a ação criar a classe `Metainfo.class` do componente *Meta-info* do SOCRATES.

O funcionamento do Digester é baseado na interface de programação de aplicação (*API – Application Programming Interface*) SAX (*Simple API for XML parsing*) [Sax05] para ler arquivos XML, por este motivo, instancia e popula os objetos Java à medida que lê o arquivo XML de entrada.

O SAX é um padrão de leitura de arquivos XML criado inicialmente para Java, mas depois estendido a outras linguagens de programação [Sax05]. O SAX representa um modelo de leitura em que o método *parse* contido na API assume todo o controle do processo empurrando objetos criados de acordo com as regras e eventos mapeados para uma pilha de instanciações de objetos, no caso do SOCRATES objetos do componente *Meta-info* e que definem toda a política de amarração desejada.

Um evento é gerado quando Digester lê os caracteres de um arquivo XML suficientes para identificar uma estrutura sintática. Por exemplo, quando o analisador sintático (*parser*) encontrar o caracter ">" no final do rótulo "<Exemplo>", irá gerar o evento início de elemento. Depois, ao reconhecer o padrão te texto "</Exemplo>" irá gerar o evento fim de elemento para o mesmo elemento [Doe04].

A seguir será analisado o Drools, que é o componente de infra-estrutura responsável por abstrair estas meta-informações e aplicar as heurísticas necessárias para simular o trabalho do especialista humano e chegar a um veredito sobre as oportunidades de refatoração.

4.1.3 Drools

Drools é uma máquina de regras (motor de inferências) baseada em linguagens declarativas que processa grafos através do algoritmo Rete-OO [Dro05] que, por sua vez, é uma extensão do algoritmo Rete [For82] original, porém, implementado em Java. Rete-OO é um algoritmo que testa um predicado declarativo contra um conjunto de fatos lógicos em tempo real. Funciona de forma análoga a uma consulta SQL que executa filtros para encontrar os resultados. Porém, diferente de uma consulta normal, o Rete-OO utiliza uma junção progressiva para ir buscando as linhas que passam pelo filtro. As linhas podem ser adicionadas a qualquer momento, montando uma visão à medida que se testa o predicado.

No caso do Drools o algoritmo Rete-OO testa fatos lógicos contra um conjunto de regras, ou seja, blocos *if/then* de código. A parte *if* testa condições sobre o domínio da

aplicação enquanto que a parte *then* executa ações diversas que são disparadas quando o bloco *if* retorna um resultado verdadeiro. Os blocos de regra são a forma mais granular de uma regra de negócios.

As regras manipuladas por Drools devem assumir o formato abaixo para serem aplicadas:

```
<rule name="NOME DA REGRA">
    <parameter identifier="PARÂMETRO DE ENTRADA">
      <java:class>CLASSE DO PARÂMETRO DE ENTRADA</java:class>
    </parameter>
    <java:condition>CONDIÇÃO DE ACESSO A REGRA</java:condition>
    <java:consequence>
        O QUE FAZER CASO A REGRA SEJA ATIVADA
    </java:consequence>
</rule>
```

Figura 4.3 – Formato da regra dentro da máquina de regras Drools.

Pode-se observar na Figura 4.3 que a regra manipulada por Drools é descrita utilizando um arquivo em formato XML, porém, os rótulos <*java:class*>, <*java:condition*> e <*java:consequence*> indicam que os trechos da regra contidos nestes rótulos serão codificados em linguagem Java. As regras de Drools podem ainda ser expressas utilizando as linguagens Groovy e Python [Dro05]. Essa flexibilidade permite que a regra incorpore algoritmos sofisticados (complementos realizados por uma aplicação externa) que podem ser utilizados na condição da regra ou no seu corpo.

As regras de Drools são conectadas à máquina de inferências através de uma estrutura chamada de *memória de trabalho*. Esta estrutura, também conhecida como espaço de fatos lógicos, é o local onde será criado o conceito de conhecimento. O conhecimento é um conjunto de fatos lógicos e representa a conexão entre as regras, ou domínio da aplicação, e a máquina de inferências. A memória de trabalho pode ser entendida como o limite entre o interior e o exterior do motor de inferências. Ela representa os limites de alcance de uma regra, uma vez que a regra é instanciada e disparada em seu interior. Observe o trecho de código da Figura 4.4.

```
1. //Inicializa um agrupamento de regras.
2. ruleBase =
   ruleBaseBuilder.buildFromUrl(this.getClass().getResource(resource));
3. //Associa a memória de trabalho com a base de regras criada.
4.workingMemory = ruleBase.newWorkingMemory();
```

Figura 4.4 – Criando uma memória de trabalho.

As linhas 2 e 4 representam a criação de uma memória de trabalho. A linha 2 cria uma base de regras de inferências a partir de um *recurso*, isto é, um arquivo de regras descrito conforme a Figura 4.3. Esta base funciona como um pacote que guarda as regras de inferência a serem utilizadas em conjunto. A linha 4 instancia a memória de trabalho, ou seja, estabelece um limite formal para uma região de inferências. Desta forma, o motor de inferências sabe que seu trabalho está finalizado quando ele completar todos os possíveis caminhos de aplicações de regras dentro de uma memória de trabalho.

Uma memória de trabalho recebe objetos externos que interagem com suas regras. Esta interação pode resultar em parâmetros das regras que podem ser usados como identificadores nas seções *condição* e *conseqüência* e em objetos auxiliares para propósitos gerais no interior do motor. Observe o trecho de programa da Figura 4.4.

```
1. workingMemory.assertObject(metainfo);
2. workingMemory.fireAllRules();
```

Figura 4.5 – Interagindo com a memória de trabalho e iniciando a execução do motor de inferências.

A linha 1 mostra como passar objetos externos à memória de trabalho e a linha 2 representa a chave de ignição do motor, após esta linha de execução o motor iniciará a aplicação de inferências.

É importante salientar que não existe uma ordem exata de aplicação das regras, apenas pontos iniciais. Para aproximar-se das análises executadas por especialistas humanos elas devem ter seu fluxo direcionado por conseqüências. Isto é, os resultados da análise de uma regra determinam qual será a próxima regra a ser analisada.

Da mesma forma acontece na análise feita por humanos; algumas situações podem aparecer ou desaparecer com o resultado de outras. Por exemplo, a refatoração *extract*

method [Fow99] acontece quando temos um método muito grande na classe, ou quando situações repetidas ocorrem no corpo do método. Mas se antes de analisarmos esta situação, trocarmos as variáveis temporárias (*replace temp with query*) [Fow99] podemos organizar o trecho de código de tal forma que a extração de método deixe de ser necessária.

Portanto, o motor de inferências é a estrutura responsável por, além de conduzir o fluxo de aplicação de regras, criar as condições necessárias para que elas funcionem corretamente. Enfim, é a infra-estrutura básica da análise que combina uma instância da classe `Metainfo.class`, que contém as meta-informações sobre o código fonte, e as *regras de identificação* de oportunidades de refatoração, produzindo como resultado um veredito sobre a necessidade de modificação do código fonte.

4.2 Componentes de negócio

SOCRATES é uma ferramenta flexível e personalizável. Em sua arquitetura foram utilizadas técnicas de Orientação a Objetos para ajudar na manutenção e organização das estruturas internas.

Estas técnicas visam aumentar a coesão interna, diminuir o acoplamento externo e organizar suas estruturas de forma que elas se apresentem divididas de forma modular em relação às suas responsabilidades. Devido ao uso das ferramentas de infra-estrutura descritas acima e ao projeto modular, os componentes de negócio do SOCRATES – *Amarração XML*, *Meta-Info*, *Análise* e *Regras de Inferência* –, apresentados a seguir, são simples e requereram quase nenhuma codificação.

A codificação necessária, no entanto, ficou concentrada no desenvolvimento dos algoritmos que implementam as *Regras de Inferência*. Eles utilizam as informações providas pelas ferramentas de infra-estrutura para a identificação das oportunidades de refatoração. O algoritmo para identificação de parâmetros obsoletos é apresentado em detalhes quando o componente *Regras de Inferência* é descrito.

4.2.1 Amarração XML (XML Binding)

SOCRATES utiliza o componente Digester para mapear o arquivo JavaML (por exemplo, o descrito na Figura 4.2) em classes do componente *Meta-info*. A seguir são descritos os principais passos realizados durante este mapeamento.

1. Criar uma instância da classe `org.apache.commons.digester.Digester`.

```
Digester digester = new Digester();
```

2. Registrar todos os elementos da estrutura do JavaML que se configuram como padrões de texto para as conversões.

    a. Por exemplo, a instrução a seguir informa o componente Digester que existe um rótulo *<java-source-program>* no arquivo JavaML e que aninhado a ele existe outro rótulo *java-class-file*.

```
private final String FILE = "java-source-program/java-class-file";
```

3. Para cada padrão deve haver regras de processamento que são disparadas quando o mesmo é reconhecido no arquivo de entrada. Normalmente, as regras são executadas na mesma ordem em que são declaradas.

    a. A instrução a seguir informa o componente Digester que toda vez que o padrão informado pela constante `FILE` for encontrado a classe `File.class` deve ser instanciada.

```
digester.addObjectCreate(FILE,File.class);
```

    b. A instrução a seguir informa o componente Digester que toda vez que o atributo *name* for encontrado no arquivo JavaML dentro do

padrão informado o atributo *name* no objeto Java correspondente deve ser preenchido.

```
digester.addSetProperties(FILE,"name","name");
```

4. Chamar o método `digester.parse()` passando o arquivo JavaML de entrada para que a amarração seja realizada de acordo com as regras configuradas.

```
String XMLPath = "Caminho para Arquivo JavaML";
Digester.parse(XMLPath);
```

Desta forma, através do processo explicado acima, o conteúdo do arquivo JavaML extraído do programa a ser analisado pelo SOCRATES será amarrado em objetos das classes definidas pelo componente *Meta-info*. A partir desses objetos, a análise para identificação de oportunidades de refatoração pode ocorrer.

4.2.2 Meta-Info

O mapeamento do arquivo JavaML somente é possível porque existe uma estrutura de classes que o representa internamente no SOCRATES. Esta estrutura constitui o componente *Meta-Info*. Ela é extensível e foi feita para realizar os mapeamentos usados na prova de conceito descrita na Seção 4.5. A estrutura de classes de meta-informações do SOCRATES é descrita a seguir:

1. `Metainfo.class`
    1. Representa as meta-informações obtidas. Possui uma lista de arquivos (`File.class`).

2. `File.class`
    1. Representa as meta-informações contidas em um arquivo físico Java. Possui uma lista de classes (`Class.class`).

3. `Class.class`

    1. Representa as meta-informações de uma classe Java. Possui todas as estruturas internas da classe como definições/uso de métodos, variáveis, construtores, captura de exceções, lista de argumentos separada por métodos, blocos de códigos e envio de mensagens.

Para a prova de conceito realizada neste trabalho, apenas um subconjunto dos 1650 elementos de JavaML foram mapeados para classes correspondentes do componente *Meta-Info*. No entanto, o componente pode ser estendido para incluir todos os elementos.

Note-se que os componentes de infra-estrutura *Jikes* e *Digester* e os de negócio *Amarração XML* e *Meta-Info* têm a função de obter e fornecer as meta-informações necessárias à identificação de oportunidade de refatoração. Dessa maneira, os objetos instanciados da estrutura de classes do componente *Meta-Info* emulam objetos criados pelo mecanismo de reflexão da linguagem Java.

4.2.3 Análise

O componente de análise é implementado pela classe `Brain` descrita no capítulo anterior. Esta classe basicamente ativa a máquina de inferência da ferramenta Drools carregando um arquivo XML de Regras em uma dada memória de trabalho. Este arquivo descreve as condições que ativarão algoritmos (Regras de Inferência) para verificar a ocorrência de uma oportunidade de refatoração.

Arquivo XML de regras

Observe-se na Figura 4.6 novamente o arquivo XML de regras para identificação de parâmetro obsoleto.

```
1: <rule name="parametro obsoleto">
2:    <parameter identifier="metainfo">
      3:<java:class>socrates.metainfo.java_source_program.Metainfo</java:class>
4:    </parameter>
5:    <java:condition>metainfo != null</java:condition>
6:    <java:consequence>
7:        socrates.rules.obsoleteparameter.ObsoleteParameter ob = new
socrates.rules.obsoleteparameter.ObsoleteParameter(metainfo);

8:         java.util.List obsoleteParamenters = ob.getObsoleteParameters();

9:           if(!obsoleteParamenters.isEmpty()){
10:              System.out.println("Os seguintes parametros sao obsoletos: ");
11:              for (java.util.Iterator iObsolete =
obsoleteParamenters.iterator(); iObsolete.hasNext();) {

12:socrates.metainfo.java_source_program.java_class_file.class_decl.method.Argume
nt argument =
(socrates.metainfo.java_source_program.java_class_file.class_decl.method.Argument
) iObsolete.next();
13:System.out.println("* "+argument.getName());
14:            }
15:          }else{
16:                System.out.println("Não existem parametros obsoletos");
17:          }
18:    </java:consequence>
19: </rule>
```

Figura 4.6 –Arquivo de regras XML do componente Drools para identificação de parâmetros obsoletos.

A regra descrita no arquivo XML é dividida em duas partes. O núcleo/inteligência e o protocolo. O protocolo é a condição necessária para ativar o motor de inferências e seus recursos, conforme visto na Figura 4.3. A inteligência é uma classe Java (`socrates.rules.obsoleteparameter.ObsoleteParameter`) que implementa um algoritmo que utiliza os dados contidos em objetos da classe `Metainfo.class` para identificar oportunidades de refatoração. Esta classe é ativada dentro do rótulo *<java:consequence>* quando é criado um objeto (linha 7) que recebe como parâmetro um objeto da classe `Metainfo.class`.

Na linha 8, o método `getObsoleteParameters()` aplica o algoritmo de identificação de parâmetros obsoletos no objeto da classe `Metainfo.class` e retorna uma lista de parâmetros. As linhas 9 a 17 contêm código Java para imprimir o resultado da análise do código.

Divisão de responsabilidades

Dessa forma é promovida uma divisão de responsabilidades em que o motor de inferências se ocupa em cuidar do fluxo de execução das regras, conforme definido em seu protocolo, e o núcleo (Regras de Inferências) em executar a análise de identificação propriamente dita das oportunidades de refatoração.

Observe-se o caso da identificação de parâmetros obsoletos. Para conseguir descobrir um parâmetro obsoleto em um conjunto de classes devemos analisar se este parâmetro é utilizado na classe corrente e em qualquer uma das subclasses que sobrescrevem o método que contém o parâmetro analisado. Esta investigação deve ser repetida para todos os parâmetros do método e para todos os métodos de todas as classes do programa. O algoritmo para identificação do parâmetro obsoleto (descrito na Seção 4.2.2) é relativamente complexo para ser desenvolvido dentro de um arquivo XML por três razões.

- Não existe um ambiente de desenvolvimento (*IDE – Integrated Development Envirement*) para a geração do arquivo XML com trechos de programas escritos em Java.

- O algoritmo não poderia ser utilizado fora da regra. Até mesmo seu compartilhamento entre diversas regras seria comprometido.

- O arquivo XML ficaria muito grande o que comprometeria a manutenção das regras e o mapeamento de seu fluxo dentro do motor de inferências.

A ativação da regra de inferência no arquivo XML de regras da Figura 4.6 chama uma classe externa que executa sua inteligência, ou seja, descobrir se existe ou não um parâmetro obsoleto. Assim, o protocolo é utilizado pelo motor de inferências para determinar o fluxo de aplicação das classes que identificam oportunidades de refatoração (no exemplo acima há apenas uma), além de suas interações com outras regras no arquivo XML de Regras do Drools, além de outros elementos externos a memória de trabalho.

Se a opção fosse por consultar as regras diretamente no formato Java, não respeitando o protocolo de comunicação com o motor de inferências, seria necessário implementar toda a capacidade do componente de análise de agir como um sistema especialista. Ou seja, seria necessário implementar código para interagir com objetos externos e internos ao contexto das regras, para guardar estados das informações entre as execuções das regras, para mudar o fluxo de aplicação de acordo com os resultados prévios obtidos, para aprender situações vividas pelo sistema, para otimizar fluxos internos e para representar os conceitos de inteligência artificial utilizados.

4.2.1. Regras de inferência

O componente *Regras de Inferência* é composto pelas classes que implementam os algoritmos para a identificação de oportunidades de refatoração. Estas classes, chamadas simplesmente de *regras de inferência*, são a alma da análise. Elas concentram quase toda inteligência usada na representação do trabalho analítico do especialista que estaria realizando esta atividade. Possuem para o SOCRATES, entretanto, uma importância ainda maior.

Além de sua função óbvia de identificar oportunidades de refatoração, as *regras de inferência* são a documentação do que deve ser realizado. Desta forma, podem servir de base de conhecimento para aprendizagem do ofício de análise de código e, assim, assumir um papel similar a um padrão de projeto [Gam00] no que tange à transferência de conhecimento dentro da equipe e elaboração de um vocabulário da cultura corporativa.

Identificação da oportunidade de refatoração parâmetro obsoleto

Para a prova de conceito realizada com SOCRATES, foi implementada a regra de inferência para identificação de parâmetros obsoletos. A classe que implementa esta *regra de inferência* chama-se `socrates.rules.obsoleteparameter.ObsoletePara-meter` e está descrita no Apêndice A. Abaixo é apresentado o algoritmo de identificação de

parâmetros obsoletos em pseudo-código; este algoritmo está implementado no `getObsoleteParameters()`.

```
Passo 1 – Armazenar todas as classes do sistema a ser analisado em uma lista
Passo 2 –   Para cada classe verificar todos os métodos
Passo 3 –     Para cada método verificar todos os parâmetros
Passo 4 –       Para cada parâmetro
Passo 5 –       Verificar se é utilizado no bloco de código local do método
Passo 6 –       Se o parâmetro for utilizado
Passo 7 –         Retornar Passo 4 // Conclusão – Este parâmetro não é obsoleto,
verifica próximo
Passo 8 –       Se o parâmetro não for utilizado
Passo 9 –         Buscar todas as subclasses diretas (no primeiro nível
imediatamente abaixo) da classe corrente
Passo 10 –        Para cada subclasse,
Passo 11 –          Verificar se o método foi sobrescrito
Passo 12 –          Se o método foi sobrescrito
Passo 13 –            Verificar se o parâmetro foi utilizado no bloco de código
local do método na subclasse, em caso afirmativo: repetir Passo 4 // Conclusão –
Este parâmetro não é obsoleto, verifica próximo
Passo 14 –            Se o parâmetro não foi utilizado no bloco de código local
do método na subclasse, repetir Passo 7 // Conclusão – Ainda existe a chance de
ser utilizado
Passo 15 –          Se o método não foi sobrescrito
Passo 16 –            Repetir Passo 7 // Conclusão – Ainda existe a chance de ser
utilizado
Passo 17 –        Se classe não possui subclasse
Passo 18 –          Retornar Verdadeiro // O parâmetro é obsoleto
Passo 19 –        Se lista de subclasses vazia
Passo 20 – Retornar Verdadeiro // O parâmetro é obsoleto
```

Figura 4.7 –Algoritmo utilizado na regra de inferência para identificar parâmetros obsoletos.

4.3 Inclusão de novas oportunidades de refatoração no SOCRATES

Para incluir a identificação de novas oportunidades de refatoração, dois passos básicos devem ser seguidos: implementar a *regra de inferência*; e inseri-la no componente *Análise*. O primeiro passo consiste em implementar o algoritmo que representa a *regra de inferência* para identificação da nova oportunidade de refatoração. Este algoritmo, em geral, é implementado por uma ou mais classes Java responsáveis por emular o raciocínio de um especialista humano para avaliar o código de um programa e encontrar um tipo de problema específico durante a fase de análise realizada pelo SOCRATES. O segundo passo requer criar o arquivo XML de regras utilizado pelo componente de infra-estrutura Drools que irá ativar a nova regra de inferência.

Portanto, a complexidade de se identificar a oportunidades de refatoração no processo de análise do SOCRATES corresponde às dificuldades de se configurar o fluxo de execução das regras através dos arquivos XML de Regras do Drools, descrever um algoritmo que capte todo o processo que seria executado por um especialista e a tradução deste algoritmo para o programa Java.

Quando o conjunto de *regras de inferências* incluídas no SOCRATES crescer, o entendimento do funcionamento dos arquivos XML de regras pode se tornar uma tarefa penosa caso eles não sejam organizados de forma adequada. Para isso, pode-se criar um arquivo XML de regras mestre (geral) que inclui arquivos de regras para cada assunto específico que se quer analisar. Assim, o arquivo mestre define todas as regras utilizadas na análise e os arquivos incluídos definem subfluxos específicos do fluxo invocado pelo arquivo mestre.

Depois de configurado o fluxo de execução das regras, verificar se o arquivo criado já é um subfluxo de algum fluxo previamente configurado, conforme explicado acima. Se este for o caso, as regras chamadas de dentro deste novo subfluxo já serão automaticamente consideradas pela análise do SOCRATES.

Caso exista a necessidade de substituição total dos fluxos previamente configurados, basta passar o arquivo drl (extensão Drools) que contém o fluxo principal como parâmetro para o método `setup` da classe `Brain.Java` descrito na Figura 4.7. Por exemplo, supondo que um novo arquivo de XML de regras "`rules.java.drl`" seja criado, para que ele seja ativado no SOCRATES, basta que o método `setup` receba a cadeia "`rules.java.drl`" para que seja ativado.

4.4 Alterando os Componentes de Negócio do SOCRATES

Além da inclusão da identificação de novas oportunidades de refatoração, o SOCRATES pode também ter seus componentes de negócios alterados, desde que a relação de dependências interna seja respeitada.

De acordo relação de dependências listada na Figura 6.2, a alteração de um componente pode implicar a manutenção de outros. Desta forma, se qualquer um dos componentes de negócio não estiver funcionando de acordo com o esperado, eles podem ser alterados e até mesmo substituídos.

De acordo com a Tabela 4.1, o componente *Amarração XML* depende do componente *Meta-info*, isso significa que se este componente (Meta-info) for alterado, o componente *Amarração XML* será afetado.

Ter dependência não significa que o impacto no componente afetado, neste caso, *Amarração XML*, impeça a alteração. Em todos os componentes, os pontos de dependências externas são, normalmente, localizados em uma classe de interface/limite entre os componentes e desta forma o impacto é reduzido e não difuso dentro do componente. A mesma análise se repete para todos os componentes listados na Tabela 4.1.

Tabela 4.1 – Relação de dependências entre os componentes internos do SOCRATES.

Componente	**Amarração XML**	**Meta-info**	**Análise**	**Regras**
Amarração XML	----	Sim	Não	Não
Meta-info	Não	----	Não	Não
Análise	Não	Sim	----	Sim
Regras	Não	Sim	Não	----

4.5 Exemplo de utilização do SOCRATES

O SOCRATES é um protótipo construído como prova de conceito para o processo de identificação de oportunidades de refatoração. Para avaliar suas funcionalidades foi aplicado a um exemplo mais complexo que o descrito no Capítulo 3. O exemplo consiste na análise de uma classe utilizada no desenvolvimento do próprio SOCRATES. A classe `Master.java` (Figura 4.8) é utilizada para iniciar o processo de análise do SOCRATES e será analisada pelo próprio SOCRATES.

```
1   package execucao;
2
3   import socrates.Brain;
4   import socrates.metainfo.java_source_program.Metainfo;
5   import socrates.xml.extractor.MetainfoExtractor;
6
7   /**
 8   * @author André Piza (pizzandre@hotmail.com)
 9   *
 10 * Arquivo mestre. Chama o motor de inferencias e o executa.
 11 */
12  public class Master {
13
14      public Master(){
15          MetainfoExtractor extractor;
16
17             try {
18
19                 extractor = new MetainfoExtractor("file:C:\\Documents and
Settings\\Usuário\\Meus 20documentos\\Mestrado\\Exemplos\\estudo de
caso\\socrates\\etc\\xml-unparsed\\Master.java.xml");
21             Metainfo metainfo = extractor.parseClass();
22
23
24                   Brain brain = new Brain();
25                   brain.activate(metainfo);
26
27             } catch (Exception e) {
28                   e.printStackTrace();
29             }
30  }
31
32     public static void main(String[] args) {
33           new Master();
34     }
35 }
```

Figura 4.8 – Classe Master.java utilizada para prova de conceito.

Esta classe realiza o fluxo de operações necessárias para a ativação do SOCRATES como explicado a seguir.

- Possui um método construtor na linha 14.
- Na linha 19, dentro do método construtor, é informado onde se encontra o arquivo XML correspondente ao resultado de seu processo de compilação pelo Jikes.
- Na linha 21, é criado um objeto da classe `Metainfo.java` que corresponde ao resultado do parse do arquivo XML informado acima.
- Na linha 25 é iniciado o processo de análise do próprio programa da Figura 4.8.

Esta classe também possui um método `main` com o parâmetro `args` que nunca é utilizado. O objetivo do SOCRATES é apontar este mau cheiro e, dessa maneira, indicar a possibilidade da aplicacão da refatoração *remove parameter*.

O primeiro passo do SOCRATES é a compilação do programa da Figura 4.8 com o uso do Jikes. Este passo deve ser realizado previamente. Como resultado obtém-se um arquivo XML em formato JavaML (`master.java.xml`) com as meta-informações da classe `master.java`. Ele pode ser conferido na Figura 4.9.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
<!DOCTYPE java-source-program SYSTEM "java-ml.dtd">
-->

<java-source-program>
<java-class-file name="C:/jikes/src/execucao/Master.java">
<package-decl name="execucao"/>
<import module="socrates.Brain"/>
<import module="socrates.metainfo.java_source_program.Metainfo"/>
<import module="socrates.xml.extractor.MetainfoExtractor"/>
<class name="Master" visibility="public" line="12" col="0" end-line="34" end-
col="0">
  <superclass name="Object"/>
  <constructor name="Master" visibility="public" id="Master:ctr-24" line="14"
col="8" end-line="29" end-col="3">
    <formal-arguments/>
    <block line="14" col="23" end-line="29" end-col="3">
      <local-variable name="extractor" id="Master:var-42"><type
name="MetainfoExtractor"/></local-variable>
      <try>
        <block line="17" col="20" end-line="26" end-col="16">
          <assignment-expr op="="><lvalue><var-set
name="extractor"/></lvalue><new><type
name="MetainfoExtractor"/><arguments><literal-string value="file:C:\\Documents
```

```
and Settings\\Usurio\\Meus documentos\\Mestrado\\Exemplos\\estudo de
caso\\socrates\\etc\\xml-unparsed\\Master.java.xml"/></arguments></new>
          </assignment-expr>
          <local-variable name="metainfo" id="Master:var-56"><type
name="Metainfo"/><send message="parseClass">
              <target><var-ref name="extractor"/></target>
              <arguments/>
            </send>
          </local-variable>
          <local-variable name="brain" id="Master:var-63"><type
name="Brain"/><new><type name="Brain"/><arguments/></new>
          </local-variable>
          <send message="activate">
            <target><var-ref name="brain" idref="Master:var-63"/></target>
            <arguments><var-ref name="metainfo" idref="Master:var-
56"/></arguments>
          </send>
        </block>
        <catch><formal-argument name="e" id="Master:var-74"><type
name="Exception"/></formal-argument>
          <block line="26" col="38" end-line="28" end-col="16">
            <send message="printStackTrace">
              <target><var-ref name="e" idref="Master:var-74"/></target>
              <arguments/>
            </send>
          </block>
        </catch>
      </try>
      <return/>
    </block>
  </constructor>
  <method name="main" visibility="public" static="true" id="Master:mth-34"
line="31" col="8" end-line="33" end-col="8">
    <type name="void" primitive="true"/>
    <formal-arguments>
      <formal-argument name="args" id="Master:frm-32"><type name="String"
dimensions="1"/></formal-argument>
    </formal-arguments>
    <block line="31" col="47" end-line="33" end-col="8">
      <new><type name="Master"/><arguments/></new>
    </block>
  </method>
</class>
</java-class-file>
</java-source-program>
```

Figura 4.9 – Resultado do processamento realizado pelo Jikes no programa da Figura 4.8.

O SOCRATES busca o arquivo XML que contém as metainformações e inicia o processo de análise. Esta informação é passada como parâmetro do método construtor da classe `MetainfoExtractor` (linha 19 da Figura 4.8).

Esta classe é responsável por extrair do arquivo XML uma instância da classe `Metainfo.java`, em outras palavras, ela invoca o componente de *Amarração XML* que possui o mapeamento do formato JavaML na classe `Metainfo.java`. O resultado do

processo é uma instância da classe `metainfo` com todas as informações contidas no arquivo JavaML, porém em Java, para ajudar na análise das informações pelo motor de inferências.

Neste ponto, a regra que busca por um parâmetro obsoleto já está programada e inserida em uma memória de trabalho. Assim, a combinação da memória de trabalho com a instância da classe `Metainfo.java` produzida acima gera o resultado na saída padrão do sistema conforme a Figura 4.10.

```
Os seguintes parâmetros são obsoletos:
* args
```

Figura 4.10 – Resultado do processo de análise do SOCRATES para a classe Master.java.

Como não existem outras regras para serem executadas o sistema sai do componente de análise e encerra seu processamento.

4.5 Considerações Finais

Este capítulo apresentou os componentes que formam a ferramenta SOCRATES. Os componentes de infra-estrutura, a saber, *Jikes*, *Digester* e *Drools*, formam as bases de suas operações em nível de código, enquanto que os componentes de negócios – *Amarração XML*, *Meta-Info*, *Análise* e *Regras de Inferência* – orquestram o funcionamento dos componentes de infra-estrutura e realizam a identificação de oportunidades de refatoração propriamente dita. Em conjunto, esses componentes constituem o SOCRATES e emulam o conhecimento humano na descoberta de maus cheiros em um sistema de computador.

O projeto da ferramenta SOCRATES, descrito neste capítulo, foi centrado nos seguintes aspectos:

- Utilização de ferramentas livres e disponíveis. Os componentes de infra-estrutura são derivados de ferramentas livres e realizam tarefas como análise de código e implementação das funções de sistema especialista. Dessa

maneira, foi possível obter essas funcionalidades de maneira rápida e com pouca codificação.

- Projeto modular. SOCRATES foi projetado para facilitar a modificação e a inclusão de novas regras de inferência; a codificação adicional necessária é localizada em um único ponto, restrita à codificação de regras de inferência e à alteração dos arquivos XML de Regras. As regras de inferência são implementadas em linguagem Java e podem implementar de maneira eficiente algoritmos para percorrer estruturas como a árvore sintática de um método ou classe. Assim, as regras de inferência podem ser implementadas de maneira eficiente visto que a detecção de mau cheiros requerem consultas ao código.
- Projeto extensível. Na presente versão do SOCRATES, apenas o mau cheiro *parâmetro obsoleto* está implementado. Porém, o sistema baseado em regras utilizado no SOCRATES permite a inclusão de novas regras de inferências e a combinação dos resultados dessas regras para a detecção de novos mau cheiros ou a própria aplicação de refatorações.
- Constância do processo de análise automática. O processo estabelecido no SOCRATES é seguido sem exceções. Assim, muitos erros podem ser prevenidos. No caso de humanos existem sentimentos e sensações externas que podem influenciar e desviar o processo de análise. Com um programa de computador isto não acontece, pois as regras são tangíveis assim como os parâmetros que as influenciam.

Com o objetivo de exercitar os conceitos apresentados, o SOCRATES foi aplicado em um exemplo. O exemplo consistiu da análise e busca da oportunidade de refatoração *parâmetro obsoleto* em uma das classes internas do SOCRATES.

Capítulo 5

Conclusões

5.1 Síntese do Trabalho

Refatoração, segundo Fowler e colegas [Fow99], é "uma mudança interna na estrutura do software a fim de que ele se torne mais fácil de entender e de modificação mais barata, sem, contudo, modificar seu comportamento observável". De acordo com estes autores, um conjunto de refatorações aplicadas pode melhorar a manutenibilidade e a qualidade do software.

Porém, identificar oportunidades de refatoração manualmente é trabalhoso e sujeito a erros. Isto porque esta tarefa tem como pré-requisito a compreensão do software. Estima-se que cerca de 50% do tempo gasto em manutenção seja dispendido compreendendo o código fonte para se descobrir quais modificações são necessárias. Os pontos de modificação nem sempre são fáceis de ser encontrados e, às vezes, é necessário um estudo profundo do software para poder identificar uma pequena necessidade de modificação.

Segundo Tourwé e Mens [Tou03], a identificação de oportunidades de refatoração e a aplicação de refatorações manuais apresentam as seguintes dificuldades:

- Visão local do código fonte da aplicação. É difícil prever os efeitos da refatoração para a aplicação como um todo visto que o programador possui uma visão local do código.

- Dificuldade para realizar consultas sofisticadas ao código fonte. A identificação de oportunidades de refatoração requer consultas sofisticadas (e.g. *encontrar todos os métodos que acessam uma particular variável de instância*), enquanto que os ambientes de desenvolvimento atuais permitem apenas executar consultas pré-definidas (e.g., *todos os usuários de uma classe*).

- Documentação não confiável e desatualizada.

- Desconhecimento sobre refatoração. Mesmo que os desenvolvedores estejam a par dos problemas que podem levar à degradação do projeto ou à violação de convenções, muitas vezes eles não conhecem as técnicas de refatoração ou têm dificuldade para decidir qual refatoração aplicar.

Neste trabalho final foi apresentado o SOCRATES – Sistema Orientado para Caracterização de Refatorações. O objetivo principal do SOCRATES é identificar de maneira automática oportunidades de refatoração e, dessa forma, a reduzir o esforço associado à aplicação de refatorações.

Além desse objetivo principal, o projeto e a arquitetura de SOCRATES tem o objetivo de serem *leves*. Leves no sentido de reduzir a codificação necessária utilizando ferramentas livres e disponíveis e também no sentido de permitir que os algoritmos de análise de código sejam implementados de maneira eficiente. A versão de SOCRATES apresentada neste trabalho identifica o mau cheiro *parâmetro obsoleto* utilizando estes conceitos de projeto e de arquitetura.

Para atingir estes objetivos, SOCRATES utiliza componentes de infra-estrutura disponíveis em ferramentas livres e componentes de negócio especialmente desenvolvidos. A seguir, os componentes de infra-estrutura – *Jikes*, *Digester* e *Drools*, bem como os de negócio – *Amarração XML*, *Meta-Info*, *Análise* e *Regras de Inferência*, são descritos brevemente:

- *Jikes* é um compilador estendido que juntamente com *bytecodes* produz um arquivo XML que representa as meta-informações (e.g., chamadas de objetos, definições de classes, métodos, variáveis e parâmetros) do programa compilado. Este arquivo XML é baseado no padrão JavaML criado por Badros [Bad00]. Este formato foi criado para facilitar o estudo da estrutura de programas escritos em Java por ferramentas de engenharia de software.

- *Digester* é um *framework* que mapeia os arquivos XML em uma estrutura de objetos.

- *Drools* é a máquina de regras utilizada no SOCRATES que ativa a aplicação algoritmos de análise do código (Regras de Inferência). As regras utilizadas pelo *Drools* são especificadas também em arquivos XML.

- *Amarração XML* é o componente de negócios que controla o *Digester* de forma a mapear o conteúdo do arquivo JavaML nos objetos de meta-informações internos ao SOCRATES.

- *Meta-Info* é a estrutura de objetos internos da ferramenta que contém as meta-informações do código a ser analisado pelo SOCRATES. A informação contida neste componente é semelhante à fornecida pelo mecanismo de reflexão das linguagens orientadas a objetos.

- O componente de *Análise* é responsável por ativar os algoritmos de análise (Regras de Inferência) na ordem correta e com as informações necessárias para análise. Realiza este processo ativando o sistema de regras do *Drools*.

- *Regras de Inferência* são os algoritmos de análise propriamente ditos. Eles não fazem parte da estrutura do *Drools*, mas são ativados por suas regras. As Regras de Inferências são escritas em Java e implementam os algoritmos

para a obtenção de um veredicto sobre a existência ou não de uma oportunidade de refatoração.

Os componentes de negócio de SOCRATES requereram pouca codificação e basicamente orquestram o funcionamento dos componentes de infra-estrutura. A codificação necessária ficou concentrada nas *Regras de Inferência*. Estas regras constituem algoritmos que podem ser escritos em algumas linguagens de programação, entre elas Java. É importante observar que os algoritmos podem ser implementados de maneira eficiente para a detecção de oportunidades de refatoração. Isto não ocorre como outras ferramentas semelhantes [Tou03].

O SOCRATES foi construído tendo-se em mente que toda sua inteligência está nos algoritmos programados em Java e no fluxo de execução, ou ordem de disparo, destes algoritmos. Como decorrência da sua arquitetura, a inclusão de regras para a identificação de novas oportunidades de refatoração é facilitada. Basicamente, isto requer a codificação dos algoritmos de análise (Regras de Inferência) e o estabelecimento das condições para sua ativação pelo *Drools*. É importante notar que utilizando esta solução é possível combinar diferentes algoritmos para a tomada de decisão sobre a existência ou não de uma oportunidade de refatoração. Isto pode ser útil para decidir sobre a aplicação de refatorações em cascata como sugerido por Mens e Tourwé [Tou03].

.

Desta maneira, o presente trabalho mostrou que é possível construir, utilizando ferramentas livres e disponíveis, um sistema de identificação automática de mau cheiros. Além disso, este sistema requer pouca codificação que pode ser implementada de maneira eficiente. Portanto, é possível observar que SOCRATES é um sistema orientado a objetos – pois utiliza conceitos de projeto e arquitetura e ferramentas disponíveis neste paradigma – para a caracterização de refatorações, isto é, para identificar oportunidades de refatoração.

5.2 Contribuições e Trabalhos Futuros

A principal contribuição do SOCRATES é provar o conceito de que é possível construir uma ferramenta eficiente para identificar maus cheiros com pouca codificação localizada utilizando ferramentas livres e disponíveis. A identificação do mau cheiro *parâmetro obsoleto* mostra que o projeto e a arquitetura utilizada é factível.

No entanto, a arquitetura proposta e a primeira versão do SOCRATES incentivam a realização de alguns trabalhos futuros, a saber:

- Estudo de caso com programas de maior porte. A presente versão do SOCRATES foi avaliada apenas para provar o conceito. No entanto, é importante verificar a eficácia do algoritmo implementado para detecção de *parâmetros obsoletos* em sistemas como muitas classes e métodos.

- Avaliação de desempenho. SOCRATES permite que os algoritmos de análise de código sejam implementados de maneira eficiente. Mas é necessário avaliar o tempo necessário para detectar *parâmetros obsoletos* em programas reais.

- Inclusão de novas oportunidades de refatoração. Nem todo mau cheiro é factível de ser implementado de forma algorítmica, porém, é importante implementar no SOCRATES a identificação de outras oportunidades de refatoração que possuem soluções algorítmicas, por exemplo, o mau cheiro *interface obsoleta*.

- Integração com métricas. O resultado das Regras de Inferências pode ser combinado com o uso de métricas para a identificação de oportunidades de refatoração. A arquitetura do SOCRATES, que utiliza a máquina de regras do Drools, facilita a integração de soluções algorítmicas e soluções baseadas em métricas.

- Aplicação automática de refatoração. O SOCRATES realiza a identificação de maus cheiros, que é um dos passos para a aplicação de refatoração. Porém, o sistema pode também conter facilidades para a aplicação automática de refatorações.

- Aplicação de refatorações em cascata. Após a aplicação de uma refatoração, outras podem ser aplicadas em seguida. O sistema de regras do SOCRATES baseado no *Drools* pode ser utilizado para permitir a aplicação de refatorações em cascata.

- Integração das ferramentas. Os componentes do SOCRATES não estão integrados. Um trabalho importante é integrar estes componentes em uma interface gráfica ou mesmo como um *plugin* de uma ambiente de desenvolvimento.

Referências Bibliográficas

[Bad00] G. Badros. "JavaML: a markup language for Java source code", Computer Networks, 33(1-6), june, 2000.

[Bec99] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999.

[Car03] G. F. Carneiro; M. G. Mendonça Neto. "Relacionando *Refactorings* e Métricas de Código Fonte – Um Primeiro Passo Para A Detecção Automática De Oportunidades De *Refactoring*". Anais do 17º. Simpósio Brasileiro de Engenharia de Software, Manaus, AM, 2003.

[Clo94] W. F. Clocksin; C. S. Mellish. *Programming in Prolog*. Berlin, Alemanha: Springer-Verlag, 1994.

[Dig05] Digester. http://jakarta.apache.org/commons/digester/.Visitado em 11 de outubro de 2005.

[Doe04] O. Doederlein "A Dinâmica de Java Reflection e Meta-programação". JavaMagazine ed 14, ano II, pág. 38-50, Abril/2005.

[Doe05] O. Doederlein, "XML Turbinado com Java", JavaMagazine ed 22, ano III, pág. 36-48, Março/2005.

[DrD05] http://drools.org/Declarative+Programming. Visitado e 11 de outubro de 2005.

[Dro05] http://drools.org/. Visitado em 11 de outubro de 2005.

[Drfaq06] http://drools.codehaus.org/FAQ?nocache. Visitado em 15 de Janeiro de 2006.

[Ecl05] Eclipse. http://www.eclipse.org/. Visitado em 11 de outubro de 2005.

[For82] C. Forgy. "Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem", Artif. Intell. 19(1): 17-37 (1982).

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional , 1999. 431 p.

[Fow05] M. Fowler. http://www.refactoring.com/rejectedExample.pdf. Visitado em 04 de janeiro de 2006.

[Gam05] E. Gamma; K. Beck. “JUnit Testing Infected: Programmer love writing tests”, http://junit.sourceforge.net/doc/testinfected/testing.htm. Visitado em 04 de janeiro de 2006.

[Gam00] E. Gamma; R. Helm; R. Johnson; J. Vlissides. *Padrões de Projeto – Soluções Reutilizáveis de Software Orientado a Objetos*. Porto Alegre: Bookman, 2000.

[Gol89] A. Goldberg; D. Robson. *Smalltalk-80 The Language*. Reading, MA, EUA: Addison-Wesley, 1989.

[Gre05] D. Green. http://java.sun.com/docs/books/tutorial/reflect/. The Reflection API, visitado em Outubro de 2005.

[Gur01] J. Gurp and J. Bosch. “Design Erosion: Problem & Causes”, Journal of System & Software, 61(2):105-119, 2001.

[Har01] E.R Harold; W.S. Means. XML in a nutshell: a desktop quick reference. Beijing: O' Reilly, 2001.

[Jcp04] JSR-94. http://www.jcp.org/en/jsr/detail?id=94. JSR 94: Java Rule Engine API, final release. 04/08/2004.

[Jik05] Jikes. http://jikes.sourceforge.net/. Visitado em 05 de setembro de 2005.

[May96] A. Mayrhauser; A. Mary Vans. “Program Comprehension During Software Maintenance an Evolution”, IEEE Computer, 28(8) pp. 44-45,1996.

[McM97] McManis, Chuck. http://www.javaworld.com/javaworld/jw-09-1997/jw-09-indepth.html. “Take an in-depth look at the Java Reflection API”. JavaWorld Java in deph tutorials, Setembro/1997.

[Men04] T. Mens e T. Tourwé, “A Survey of Software Refactoring”, IEEE Transactions on Software Engineering, 30(2): 126-139.

[Opd92] W. F. Opdyke, *Refactoring Object-oriented Frameworks*. Ph.D thesis, University of Illinois at Urbana-Champaign, 1992.

[Pig97] T. M. Pigoski, *Practical software maintenance: best practices for managing your software investment*. New York: John Wiley, 1997. 384p.

[Piz04] A. Piza. http://cvs.atech.br/framework/components/decision.html. Decision, Vidatis-Framework. Novembro 2004.

[Sax05] http://www.saxproject.org. Visitado em 19 de outubro de 2005.

[Tou03] T. Tourwé; T. Mens, “Identifying Refactoring Opportunities Using Logic Meta Programming”, Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR ’03).

Apêndice A

Código fonte do programa Java para parâmetro obsoleto

```
package socrates.rules.obsoleteparameter;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;

import socrates.metainfo.java_source_program.Metainfo;
import socrates.metainfo.java_source_program.java_class_file.Class;
import socrates.metainfo.java_source_program.java_class_file.File;
import
socrates.metainfo.java_source_program.java_class_file.class_decl.method.Argument;
import
socrates.metainfo.java_source_program.java_class_file.class_decl.method.Block;
import
socrates.metainfo.java_source_program.java_class_file.class_decl.method.Construct
or;
import
socrates.metainfo.java_source_program.java_class_file.class_decl.method.LocalVari
able;
import
socrates.metainfo.java_source_program.java_class_file.class_decl.method.Method;
import
socrates.metainfo.java_source_program.java_class_file.class_decl.method.MethodHan
dler;

/**
 * @author André Piza (pizzandre@homail.com)
 * Classe responsável por mapear as regras necessárias para se encontrar o
<i>"bad smell"</i>
 * <code>parâmetro obsoleto</code>.
 */
public class ObsoleteParameter {

    /** Propriedade <code>metaInfo</code> do tipo <code>Metainfo</code> */
    private Metainfo metaInfo = null;

    /** Propriedade <code>subclassInfo</code> do tipo <code>HashMap</code> */
    private HashMap subclassInfo = new HashMap();
```

```
    /** Propriedade <code>classInfo</code> do tipo <code>HashMap</code> */
    private List classInfo = new ArrayList();

    /** Propriedade <code>obsoleteParameters</code> do tipo <code>List</code> */
    private List obsoleteParameters = null;

    /** Método construtor da classe <code>ObsoleteParamenter</code> */
    public ObsoleteParameter(Metainfo metaInfo) {
        super();
        this.obsoleteParameters = new ArrayList();
        this.metaInfo = metaInfo;
        this.activate();
    }

    /**
     * Método que ativa a regra do parâmetro obsoleto.
     */
    private void activate() {
        if (this.metaInfo == null) {
            return;
        }

        for (Iterator iFile = metaInfo.getFiles().iterator(); iFile.hasNext();) {

            /*
             * Busca todos os arquivos parseados
             */
            File file = (File) iFile.next();

            /*
             * Adiciona a classe na lista de classes parseadas.
             */
            classInfo.add(file.getClassMetaInfo());

            /*
             * Estrutura de armazenamento das sub-classes
             * 1-Verifica se existe alguma classe com o nome da super-classe
armazenado;
             * 2-Adiciona o valor classe, tendo como chave o nome da subclasse;
             * 3-Todas as classes ficam armazenadas, agrupadas pelo nome da
super-classe;
             */
            List metaInfoList = (List) subclassInfo.get(file.getClassMetaInfo()
                    .getSuperclass().getName());
            if (metaInfoList == null) {
                metaInfoList = new ArrayList();
            }
            metaInfoList.add(file.getClassMetaInfo());
            subclassInfo.put(file.getClassMetaInfo().getSuperclass().getName(),
                    metaInfoList);
        }

        /*
         * Forma de recuperação
         * 1-Percorrer a lista de classes;
         * 2-Verifica estruturas da classe;
         * 3-Com o nome da classe, buscar as outras informações:
         *   3.1-sub-classes;
         */
        for (Iterator iClasses = classInfo.iterator(); iClasses.hasNext();) {
            /*
             * Procura parâmetros obsoletos dentro da classe
             * (métodos de negócio e construtores). Este método
```

```
             * alimenta a lista de parâmetros obsoletos.
             */
            Class classMetaInfo = (Class)iClasses.next();
            this.searchObsoleteParamenterInClass(classMetaInfo,
this.obsoleteParameters);

            /*
             * Verifica se os argumentos obsoletos na classe em análise, também
são
             * obsoletos nas suas sub-classes;
             */
            if(!this.obsoleteParameters.isEmpty()){
                List subClasses =
(List)subclassInfo.get(classMetaInfo.getName());
                if(subClasses != null){
                    for (Iterator iSubClass = subClasses.iterator();
iSubClass.hasNext();) {
                        Class subClassMetaInfo = (Class) iSubClass.next();

this.searchObsoleteParamenterInSubClass(subClassMetaInfo);
                    }
                }

            }
        }
    }

    /**
     * Procura pelos parâmetros obsoletos na classe especificada.
     *
     * @param classMetaInfo As meta informações da classe que se quer buscar o
     * parâmetro obsoleto.
     */
    private void searchObsoleteParamenterInClass(Class classMetaInfo, List
obsoleteParameters) {
        /*
         * Procura argumentos dos métodos construtores;
         */
        List constructors = classMetaInfo.getConstructors();
        for (Iterator iConstructors = constructors.iterator();
iConstructors.hasNext();) {
            Constructor constructor = (Constructor) iConstructors.next();
            obsoleteParameters.addAll(this.searchObsoleteArguments(constructor));

        }
        /*
         * Procura argumentos dos métodos de negócio;
         */
        List methods = classMetaInfo.getMethods();
        for (Iterator iMethod = methods.iterator(); iMethod.hasNext();) {
            Method method = (Method) iMethod.next();
            obsoleteParameters.addAll(this.searchObsoleteArguments(method));

        }
    }

    /**
     * Procura por parâmetros obsoletos nas subclasses. Porém, restringe a busca
     * aos parâmetros encontrados na classe top-level da análise.
     * Altera os valores da variável <code>this.obsoleteParameters</code>.
     * @param classMetaInfo As meta-informações da classe que se quer buscar as
     * informações.
     */
```

```
    private void searchObsoleteParamenterInSubClass(Class classMetaInfo) {
        if(this.obsoleteParameters != null &&
!this.obsoleteParameters.isEmpty()){

            List obsoleteInSub = new ArrayList();

            searchObsoleteParamenterInClass(classMetaInfo, obsoleteInSub);

            /*
             * Retira os parâmetros que não são obsoletos na sub-classe
             */
            for (Iterator iObsoleteInSub = obsoleteInSub.iterator();
iObsoleteInSub.hasNext();) {
                Argument obsoleteArgInSub = (Argument) iObsoleteInSub.next();
                if(!this.obsoleteParameters.contains(obsoleteArgInSub)){
                    int index =
this.obsoleteParameters.indexOf(obsoleteArgInSub);
                    while(index >= 0){
                        this.obsoleteParameters.remove(index);
                        index =
this.obsoleteParameters.indexOf(obsoleteArgInSub);
                    }
                }
            }
        }
    }

    /**
     * Procura por argumentos não ustilizados no método/construtor
     * passado como parâmetro.
     *
     * @param handler O manipulador de estruturas para o método/Construtor
     * @return Retorna a lista de argumentos obsoletos no método.
     */
    private List searchObsoleteArguments(MethodHandler handler){

        List result = new ArrayList();
        List argumentList = handler.getArgumentList();
        List blockList = handler.getBlockList();
        Comparator searchArgInBlock = new CompareBlockList();

        for (Iterator iArgument = argumentList.iterator(); iArgument.hasNext();)
{
            Argument argument = (Argument) iArgument.next();
            //Só coloca no resultado os parâmetros obsoletos;
            if(Collections.binarySearch(blockList, argument, searchArgInBlock) <
0){
                /**@todo a busca naum foi totalmente testada*/
                result.add(argument);
            }

        }
        return result;
    }

    /**
     * Compara uma lista de blocos de código para saber se o argumento está sendo
usado
     * por ele.
     * @author André Piza (pizzandre@homail.com)
     */
    private class CompareBlockList implements Comparator{
        /**
```

```
         * Retorna 0 quando encontrar o argumento sendo utilizado por uma
         * variável local em um bloco de código. Considera os blocos de código
         * aninhados recursivamente.
         */
        public int compare (Object source, Object key){
            Block block = (Block)source;
            Argument argument = (Argument)key;
            String id = argument.getId();

            if(block.getBlockList() != null && !block.getBlockList().isEmpty()){
                for (Iterator iBlock = block.getBlockList().iterator();
iBlock.hasNext();) {
                    Block blockItem = (Block) iBlock.next();
                    if(compare(blockItem,id) == 0){
                        return 0;
                    }
                }
            }

            return verifyLocalVariableInBlock(id, block);
        }

        /**
         * Método responsável por
         *
         * @param id
         * @param localVariables
         * @return
         */
        private int verifyLocalVariableInBlock(String id, Block block) {
            List localVariables = block.getLocalVariables();
            for (Iterator iLocalVariable = localVariables.iterator();
iLocalVariable.hasNext();) {
                LocalVariable localVariable = (LocalVariable)
iLocalVariable.next();
                if(id.equals(localVariable.getId())){
                    return 0;
                }
            }
            return 1;
        }
    }

    /**
     * Método utilizado para recuperar o valor da
     * variável obsoleteParameters
     * @return Retorna os parâmetros obsoletos encontrados.
     */
    public List getObsoleteParameters() {
        return obsoleteParameters;
    }
}
```