DEPARTAMENTO DE COMPUTAÇÃO

D E C O M

U F O P

UNIVERSIDADE FEDERAL DE OURO PRETO

UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

Ferramenta para Seleção e Aplicação de *Refactoring* Automático a Programas Java.

Eduardo Magno Lages Figueiredo

Orientador: Prof. Marcelo de Almeida Maia

Relatório Técnico DECOM

Ouro Preto - MG
Dezembro de 2003

Agradecimentos

Agradeço primeiramente a Deus pelas glórias que tenho conseguido em minha vida.

A meus pais pelo incentivo e apoio total que eles têm me dado nas decisões difíceis que tive de tomar. Meus irmãos Ângelo, Rosane e Marcelo por serem pessoas tão diferentes e tão cheias de qualidades que em muito contribuíram para formar o que eu sou. Não posso esquecer da Marina que me trás alegria mesmo em momentos tristes. Agradeço a minha vovó Chica que me ensinou os primeiros passos no mundo do conhecimento e ao Tadeu, meu tio, padrinho e exemplo de vida.

Aos professores do DECOM não tenho palavras para agradecer todo o conhecimento passado por eles. Em especial, agradeço ao professor e orientador Marcelo Maia que foi a estrutura (*framework*) deste projeto e ao professor Marcone pela motivação.

Amigos e companheiros de curso, muitas dificuldades foram enfrentadas e juntos conseguimos vencê-las. Obrigado pelas dúvidas esclarecidas e pelos momentos de descontração.

A família Arte & Manha que me acolheu em Ouro Preto, meu abraço especial. Eternamente serei grato e eternamente estarei com vocês, pois que um dia morou nesta república se torna imortalizado.

Finalmente, agradeço a universidade pública e de qualidade, materializada na Universidade Federal de Ouro Preto, que me propiciou o acesso ao vasto conhecimento acadêmico e científico pelo qual me apaixonei.

Sumário

Lista de Figuras

Lista de Exemplos de Código

Resumo e Palavras Chave

Resumo

O contexto do documento se aplica a áreas da engenharia de software que envolve reengenharia, qualidade e métricas de software. Este trabalho é dedicado ao reprojeto de programas Java (*refactoring*) e aos processos utilizados para sua aplicação. É argumentado sobre a aplicação de *refactorings* sem interferência do programador na seleção e configuração dos mesmos, ou seja, de forma totalmente automatizada. Uma ferramenta baseada em heurística de busca local é apresentada para efetuar este tipo de transformação de código.

A proposta é utilizar a aplicação automática de *refactorings* para melhor a qualidade interna do código fonte do sistema. Para isso são obtidas medidas de software que fornecem valores indicativos do nível de qualidade. Uma função de avaliação da qualidade do sistema foi elaborada com o intuito de guiar a heurística na aplicação de *refactorings*.

A conclusão obtida a partir deste trabalho é ser possível a aplicação automática de *refactorings*, porém deve-se fazer um estudo mais aprofundado a respeito da utilização deste mecanismo para melhorar a qualidade de um sistema.

Palavras Chave

Refactoring automático, métricas de software, reengenharia, Java, heurística e metaheurística.

Convenções e Terminologias

Para simplificar o entendimento, apresentam-se as seguintes definições que são válidas em todo o contexto deste trabalho.

O termo **campo** foi utilizado para variáveis de instância de uma classe, ou seja, variáveis que foram definidas no corpo da classe, mas não em um método. Quando a variável é definida em um método, esta será tratada apenas como **variável.** Já o termo **atributos** ou **características** é utilizado para referenciar tanto campos quanto métodos de uma classe.

Quanto às fontes utilizadas, elas variam das seguintes formas: a tradicional (Times New Roman) que é usada no corpo e títulos de todo o documento, variando apenas de tamanho. Para palavras em língua estrangeira, a fonte torna-se *itálico*, exceto se o termo estrangeiro também for comumente utilizado no Brasil como software. A fonte se altera para `Courier New` no código utilizado na exemplificação e quando houver comentários neste código estes estarão em *`Courier New Itálico`*.

Introdução

A aplicação abordada neste documento se enquadra em um grupo de sistemas chamados sistemas de meta-programação. Os tipos de dados básicos destes sistemas são programas ou fragmentos de programas sobre os quais são realizados operações com o objetivo de gerar novos programas. Meta-programação é uma área ampla e inclui subáreas como *pretty-printers* [Oppen, 1980], geradores de documentos [Brand & Visser, 1980], verificadores de plágio [Donaldson et al., 1981] [Wise, 1992], geradores de programas e sistemas de transformação de programas [Partsch & Steinbrüggen, 1983]. Esta última é o objeto de estudo deste projeto.

A evolução das metodologias e técnicas de desenvolvimento de software motiva um interesse na produção de um código mais legível e elegante. Tarefas como manutenção e reutilização de módulos do sistema ganham uma atenção especial durante todas as etapas de desenvolvimento. Produzir um sistema que apresente como características a facilidade de manutenção e reutilização de código não é uma tarefa trivial e pode ser impossível de ser alcançada nas primeiras versões.

"Um projeto reutilizável e flexível é difícil, senão impossível, de obter corretamente da primeira vez." [Gamma et al, 1995], p 17.

Os requisitos de um sistema evoluem de acordo com seu ciclo de vida, sendo necessários incrementos ou alterações em seu código para atender novos requisitos. Mesmo quando desenvolvedores experientes e cautelosos projetam grandes sistemas, estes apresentam possibilidades de melhorias. Em muitos casos as alterações aplicadas têm o intuito apenas de melhorar a legibilidade do código, e estas alterações que mantiverem o comportamento externo do sistema serão tratadas aqui como reprojeto ou *refactoring*.

A idéia de reprojetar um sistema tem aumentado com a proposta da *Extreme Programming (XP)* [Beck, 1999], que é uma metodologia para aperfeiçoamento da qualidade de software, e um de seus alicerces é a utilização de *refactoring*. *Refactorings*, por [Fowler, 2000], são alterações feitas à estrutura interna do software para facilitar o entendimento e permitir maior modificabilidade sem alterar sua semântica.

A aplicação de *refactoring* em um sistema pode ser feita de três formas distintas: manual, programada ou automática. Reprojetar um sistema de forma manual sem o auxilio de nenhuma ferramenta é uma tarefa complexa. Mesmo que determinada alteração possa parecer simples, o reflexo que ela pode acarretar ao restante do sistema é difícil de ser previsto. Por este motivo se justifica a utilização de ferramentas e linguagens para auxiliarem na manipulação do código.

Existem algumas ferramentas para aplicação programada de *refactoring* em programas Java disponíveis no mercado, sendo as mais conhecidas e analisadas durante este trabalho: *JRefactory*, *JFactor*, *Together-J* e *RefactorIt*. Um problema comum apresentado por estas ferramentas é a ausência de uma forma de descrever e acrescentar novos itens, por este motivo, foi optado pela implementação de *refactorings* utilizando uma linguagem de meta-programação. Modelos de *refactorings* são propostos e catalogados por [Opdyke, 1992], [Fowler, 2000] e [Kerievsky, 2002] e a codificação nos itens deste projeto é guiada pelos modelos de Fowler.

Uma outra forma que é objeto de estudos atuais é reprojetar o sistema de forma automatizada. Por este princípio, a seleção e aplicação do *refactoring* não sofrem interferência

do programador sendo uma ferramenta responsável pela tomada de todas as decisões. Neste caso, o *refactoring* é escolhido a partir de informações obtidas do código fonte do sistema objeto.

Um dos objetivos a serem alcançados por este trabalho é verificar a possibilidade de aplicação automática de *refactoring*, guiado por uma heurística. E ao fim do projeto espera-se obter recursos para aumentar a qualidade interna de um sistema Java. A proposta é uma ferramenta para seleção e aplicação automática de *refactoring* baseada em uma heurística ou metaheurística guiada por métricas de software [Kan, 2003].

Este documento foi organizado em duas partes e nove capítulos. Na primeira parte (capítulos 1, 2, 3 e 4) é feita uma revisão bibliográfica sobre os temas relevantes do projeto e na segunda parte (capítulos 5, 6, 7, 8 e 9) são apresentadas implementações de possíveis soluções para os problemas propostos, trabalhos relacionados e conclusões sobre o projeto.

O capítulo 1 trata de qualidade e métricas de software, sendo apresentado os conceitos que motivam o interesse em melhoria da qualidade. São descritas também neste capítulo várias medidas de código fonte, dentre elas, medidas aplicadas a softwares orientados a objeto.

No segundo capitulo é feito um estudo aprofundado sobre o tema *refactoring*, sendo definido e apresentado suas principais características. São detalhados cinco dos *refactorings* propostos por Fowler: *Encapsulate Field*, *Move Field*, *Move Method*, *Extract Class* e *Inline Class*.

Técnicas de otimização baseadas em heurísticas e metaheurísticas são descritas no capítulo 3. O enfoque deste capítulo é feito sobre algoritmos de buscas locais sendo apresentado três destes algoritmos: *Hill Climbing*, *Simulated Annealing* e Busca Tabu.

Todos os aspectos de meta-programação utilizados neste projeto foram feitos utilizando uma linguagem chamada MetaJ. A estrutura desta linhagem é descrita no capítulo 4 com o intuito de familiarizar o leitor com os códigos apresentados no decorrer do documento.

No capitulo 5 é apresentado o primeiro software fruto do projeto. JSystemInfo é uma ferramenta para coleta de informações de um sistema Java e o capítulo aborda aspectos de implementação adotados nesta ferramenta.

Os *refactorings* abordados no terceiro capítulo foram implementados utilizando uma meta-linguagem (capítulo 4). O capítulo 6 apresenta a mecânica utilizada nestes *refactorings*, aspectos de implementação e os código resultante.

Uma solução para aplicação automática de *refactoring* é apresentada no capítulo 7. Neste capítulo é feito um relacionamento entre métricas de software e *refactoring*, é definida uma heurística para seleção do *refactoring* e são apresentados os aspectos de implementação da ferramenta que utiliza esta heurística.

Finalmente, no capítulo 8 são discutidos alguns trabalhos já existentes em torno do tema abordado e no capítulo 9 são apresentadas as conclusões obtidas ao término do projeto.

PARTE I: REVISÃO BIBLIOGRÁFICA

1 Métricas de Software

O desenvolvimento de grandes sistemas é uma atividade que consome tempo e pesquisa. Sabendo-se que cada etapa deste processo requer um esforço, é preciso prover informações para que a equipe tome as decisões, trace os planos, agende as atividades e aloque os recursos. Desta forma as métricas de software tornam-se necessárias para identificar onde as pesquisas são necessárias, sendo ainda uma fonte crucial para a tomada de decisões. É sabido que diversas empresas tem desenvolvido seus modelos para prever custo, qualidade e pesquisa baseada em métricas de software.

As métricas de software são divididas em métricas do produto, do processo e do projeto. As métricas do produto descrevem características do produto como tamanho, complexidade, design, performance e níveis de qualidade. Métricas de processo podem ser usadas para melhorar o desenvolvimento e manutenção do software. E as métricas de projeto descrevem as características do projeto em desenvolvimento como o número de desenvolvedores, utilização de padrões, custo, produtividade, entre outros.

Métricas de qualidade de software são um subconjunto das métricas de software. Antes de entrar mais a fundo em métricas software, é discutido na seção seguinte a qualidade de software dentro do escopo deste trabalho.

1.1 Qualidade de Software

A definição de qualidade de software não é simples, pois o termo qualidade é bastante subjetivo e também porque o software é algo intangível. Qualidade, de uma maneira geral, pode ser definida como “atendimento aos requisitos” ou “adequado para uso”. Na verdade, pode-se dizer que todo programa faz alguma coisa certa, talvez apenas não seja a coisa que queremos que ele faça.

Qualidade para o produto software requer uma definição mais específica do que as definições tradicionais. Assim, em [Kan, 2003] qualidade de software é relacionado à ausência de *bugs*, podendo ser medida pela taxa de erros (erros por milhões de linhas de código) ou pela confiabilidade (horas de operação sem falhas).

Outra definição de qualidade de software, em [Pressman, 1995], “conformidade a requisitos funcionais e de desempenho explicitamente declarados, a padrões de desenvolvimento claramente documentados e a características implícitas que são esperadas de todo software profissionalmente desenvolvido.”

Pela definição acima, podemos enfatizar um ponto importante. Há um conjunto de requisitos implícitos que freqüentemente não são mencionados (por exemplo, desejo de uma boa manutenibilidade). Se o software se adequar aos seus requisitos explícitos, mas deixar de cumprir seus requisitos implícitos, a qualidade do software será suspeita.

Várias características devem estar presentes em um software para que este seja considerado um produto de qualidade. Neste trabalho são considerados softwares orientados a objeto e a qualidade é medida em relação a algumas características. Apresenta-se a seguir descrições das características abordadas.

Manutenibilidade: esforço exigido para localizar e reparar erros num programa.

Flexibilidade: esforço exigido para modificar um programa operacional.

Testabilidade: esforço exigido para testar um programa a fim de garantir que ele execute sua função pretendida.

Reusabilidade: à medida que um programa (ou partes de um programa) pode ser reusado em outras aplicações – relacionada ao empacotamento e escopo das funções que o programa executa.

Concisão: compactação do programa em termos de linhas de código.

Expansibilidade: quanto os projetos de arquitetura, procedimentais e de dados podem ser ampliados.

Generalidade: amplitude de aplicação em potencial de componentes do programa.

Modularidade: independência funcional dos componentes do programa.

Simplicidade: quanto um programa pode ser entendido sem dificuldades.

Quanto à estrutura interna do código fonte, [Chidamber & Kemerer, 1981] apresenta algumas características que sugerem a qualidade do produto. Estas informações são apresentadas a seguir, tendo como característica a de não considerem o processo ou qualquer agente que atue na produção do software.

Complexidade: há muitas definições para complexidade de software, dois exemplos são a medida do número de métodos por classe [Chidamber & Kemerer, 1981] e a complexidade ciclomática de McCabe, que mede o número de decisões do código.

Acoplamento: número de conexões entre classes. Uma conexão entre duas classes existe se uma delas usa um método ou uma variável de instância da outra. O alto nível de acoplamento indica baixa qualidade do software.

Coesão: a coesão é definida em função de um módulo (ou classe) e indica o quanto um componente individual é necessário para o funcionamento do módulo como um todo. A coesão é proporcional a qualidade, ou seja, altos níveis de coesão de um módulo indicam uma boa qualidade deste.

Para este trabalho são relevantes as métricas de qualidade de software, sendo abordadas na seção seguinte as métricas do produto, mais especificamente as obtidas a partir do código fonte do sistema.

1.2 Métricas de Código Fonte

Muitas métricas de produto têm sido propostas por [], contudo, as métricas mais relevantes são referente ao código fonte porque este é a forma mais confiável de informação. Entretanto, as métricas de um código fonte devem ser fáceis de serem obtidas pois com diferentes versões de um software, há bastante informação para analisar. Neste capítulo abordamos as métricas de produto obtidas a partir do código fonte e que são relevantes para o trabalho.

O número de linhas de código é uma das métricas mais simples que pode ser obtida de um software. Entretanto, vários fatores podem gerar uma diferença significante no resultado da contagem. Na época da programação em assembler, onde cada linha de código correspondia a um comando, a definição de linhas de código era clara. Hoje, com as linguagens de alto nível, a contagem pode ser influenciada, dentre outras coisas pela consideração ou não de comentários, linhas em branco e/ou declarações que não são

executadas. O número de linhas de código é influenciado também pelo estilo de programação adotado.

O número de complexidade ciclomática de McCabe é uma métrica desenvolvida para avaliar a facilidade de entendimento, manutenção e teste de um sistema. É baseada na teoria clássica de número ciclomático de um grafo, ou seja, o número de regiões do grafo. Em um software esta métrica é definida pelas decisões existentes no fluxo de execução.

Número de linhas de código e complexidade ciclomática de McCabe não depende do paradigma de programação. Na seção seguinte são apresentadas outras métricas definidas para softwares orientados a objeto.

1.3 Métricas Orientadas a Objetos

A programação orientada a objeto tem crescido bastante na última década, sendo cada vez mais adotado pelas empresas de software. As métricas orientadas a objeto [Chidamber & Kemerer, 1994] [Lorenz & Kidd, 1994] foram proposta na literatura e vem sendo recentemente aperfeiçoadas. Esta seção discute algumas destas métricas relevantes ao trabalho, enfatizando que estas métricas se relacionam a qualidade e complexidade do produto, e são obtidas a partir do código fonte.

Classes, objetos, métodos, mensagens, variáveis de instância, variáveis de classe e herança são conceitos básicos da tecnologia orientada a objeto. Assim, métricas orientadas a objetos são medidas de como estas construções são usadas no processo de desenvolvimento.

Algumas métricas podem ser bastante amplas ao sistema, como por exemplo o número de subsistemas (pacotes em Java) existentes ou o número de classes em cada subsistema. Note que o número de classes pode ser generalizado para número de tipos de um subsistema, o que inclui classes e interfaces.

O número de membros de uma classe é uma medida para o tamanho da classe. Nesta contagem são considerados como membros variáveis de instancia e de classe, construtores, métodos e tipos aninhados. Também podem ser consideradas métricas para contagem de um tipo específico de membros, como número de métodos por classe ou número de variáveis de instância.

Outras métricas que são obtidas de uma classe do sistema são referentes à hierarquia. A profundidade da árvore de herança é a medida do número de níveis da hierarquia de classe até que se atinja a raiz da árvore. Na Figura 1.1 a classe `Hashtable` tem nível 2 e a classe `Dictionary` tem nível 1 em relação a raiz da árvore de herança em Java (`Object`). O número de filhos de uma classe é uma métrica que conta as subclasses imediatas de uma classe. Pela Figura 1.1, o número de filhos conhecidos da classe `Dictionary` é 1.

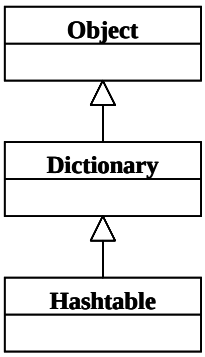

Figura 1.1: Superclasses de *Hashtable*.

A medida de acoplamento entre classes indica o número de classes que se encontram acopladas a classe em questão. Uma classe está acoplada a outra se a primeira usa métodos, variáveis de instâncias e/ou variáveis de classe definidas na segunda. Classes que possuem um alto acoplamento estão mais sujeitas à falhas do que classe de baixo acoplamento.

A coesão de uma classe é a medida de quão interligados estão os métodos de uma classe em relação ao compartilhamento de variáveis de instancia. A métrica perda de coesão da classe é obtida pelo número de pares de métodos que compartilham variáveis de instâncias, menos o número de pares de métodos que não compartilham nenhuma variável de instancia.

```
class MyClass {
 int v1, v2, v3, v4;
 void metA() { . . .  usa v1, v2  . . . }
 void metB() { . . .  usa v3  . . . }
 void metC() { . . .  usa v3, v4  . . . }
}
```

Exemplo 1.1: Classe ilustrativa de perda de coesão.

No exemplo acima há dois pares de método que não acessam nenhuma variável em comum { `metA`, `metB` } e { `metA`, `metC` }. Enquanto exatamente um par de métodos { `metB`, `metC` } compartilham a variável de instancia `v3`. A medida de perda de coesão da classe `MyClass` é 1 (2 – 1 = 1).

O tamanho de um método pode ser obtido pela contagem do número de comandos (statements) ou pelo número de linhas de código. Também se pode medir a complexidade de um método pelo número de ciclos de McCabe, discutido anteriormente.

Uma métrica interessante relacionada aos métodos de uma classe é o número de métodos definidos na superclasse e que não são redefinidos em uma subclasse. Está informação indica o reaproveitamento de código baseado em métodos reutilizados. De forma oposta, os métodos definidos na superclasse e redefinidos em uma subclasse indica uma duplicação de código indesejável.

Quanto aos campos de uma classe, um bom padrão de desenvolvimento orientado a objeto é a definição de campos com acesso privado e métodos *set* e *get* para alcançar cada campo. A medição do fator de encapsulamento de uma classe refere-se á relação entre campos de acesso privado e não privado. Esta métrica também é freqüentemente chamada de fator de ocultamento dos atributos [Harrison et al, 1998].

2 Refactoring

Manipular o código de um programa sem uma ferramenta automatizada é uma tarefa complexa, além disso, ferramentas efetuam *refactoring* de forma mais seguras e simples, reduzindo os custos de fazer software reusável.

"Com uma ferramenta, o refactoring pode ser feita de forma mais simples. Além disso, a ferramenta não comete erros tolos como eu." [Fowler, 1999] p.13

Refactoring é importante quando o desenvolvedor tiver que adicionar alguma característica ao programa, e, o código não estiver estruturado para receber esta característica. Então, primeiro é aplicado o *refactoring* no código e depois é adicionada tal característica. É importante certificar de que tenha um bom conjunto de teste antes de fazer o *refactoring*.

2.1 Definindo *Refactoring*

A proposta de efetuar *refactoring* em programas não é nova, mas vem ganhando cada vez mais força com a proposta de *Extreme Programming (XP)* [Beck, 1999]. Um dos aspectos chave da XP é a aplicação contínua de *refactoring*. Em [Roberts, 1999] afirma-se que sem *refactoring* a *Extreme Programming* não funcionará.

A forma mais comum de conceituar *refactoring* é encontrada em [Opdyke, 1992] e [Fowler, 1999]. Estes definem *refactoring* como sendo alterações feitas à estrutura interna do software que não alterem o seu comportamento observável. Apesar do conceito parecer simples este apresenta problemas em relação ao que é "comportamento observável".

Preservar o comportamento observável pode ser entendido como, o programa produzir as mesmas saídas para um mesmo conjunto de entradas, antes e após o *refactoring*. Porém, dependendo da transformação, provar que o comportamento é mantido torna-se uma tarefa árdua.

Outro problema com a definição de *refactoring* apresentada é que algumas transformações não preservam o comportamento, mas o alteram de uma maneira específica. Por exemplo, o programador pode querer substituir toda ocorrência de uma expressão pela chamada de um método que implementa uma funcionalidade similar, mas não uma funcionalidade idêntica. Este tipo de transformação também é freqüentemente tratado como *refactoring*.

Pelos problemas da definição anterior, [Roberts, 1999] propôs uma mais ampla. Este conceitua *refactoring* como sendo transformações de programas que têm um conjunto de pré-condições que devem ser satisfeitas antes de sua execução. Esta definição engloba tanto transformações que preservem ou não o comportamento do sistema.

Apesar de ambas as definições contemplarem o contexto deste trabalho e as implementações de *refactoring* que o compõe, propõe-se uma nova definição que relaciona parte das duas anteriores estritamente ao que é aqui apresentado.

Refactoring especificam transformações de programas que, atendendo um conjunto de pré-condições, garantem a preservação do comportamento do sistema ou um comportamento similar específico.

2.2 Características de um *Refactoring*

Refactoring tem como propósito melhorar a estrutura do código de forma a fazer um programa mais reusável e fácil de ser entendido, sendo então um aspecto importante para evolução de softwares reusáveis e *frameworks*. Uma característica importante é que, como o *refactoring* preserva o comportamento observável, se houver erros semânticos no código original, também haverá erros no código reestruturado. Em nossos experimentos não foi considerada a hipótese de refabricar código que apresente erros sintáticos, ou seja, foram aplicados *refactoring* apenas programas sintaticamente corretos.

Três características gerais identificadas nos *refactorings* são:

- Estes podem ser completamente automatizadas por ferramentas de *refactoring*.
- Quando um *refactoring* é corretamente aplicada, este não introduz novos erros ao sistema.
- *Refactorings* mais complexas podem ser criados pela composição de outros.
- Se cada *refactoring* primitivo preserva o comportamento do sistema, a composição destes também irá preservar.

Outras características identificadas por [Opdyke, 1992] nos *refactorings* referem-se a preservação de certas propriedades dos programas, tais como:

- Única superclasse: para C++, por exemplo, é considerada apenas herança simples.
- Classes de nomes distintos: toda classe do sistema deve ter um identificador único.
- Todo membro deve ter nomes distintos: esta propriedade obriga que os membros de uma mesma classe tenham nomes distintos. Porém, os métodos podem ser redefinidos em super ou subclasses.
- Variáveis herdadas não podem ser redefinidos: classes não podem definir variáveis que são herdadas da superclasse.
- Para redefinir, as assinaturas dos métodos devem ser compatíveis.
- Tipagem correta nas asserções: após o *refactoring*, o lado esquerdo das expressões deve ser do tipo ou de um subtipo da expressão do lado direito.
- Equivalência semântica das referências e operações: dado um conjunto de entradas, o programa deve produzir o mesmo resultado antes e após a aplicação do *refactoring*.

2.3 Exemplos de *Refactoring*

Vários padrões de *refactoring* foram catalogados por [Opdyke, 1992], [Fowler, 1999] e [Kerievsky, 2002]. Entretanto, no contexto deste documento, são tratados apenas alguns dos *refactorings* propostos por Fowler com os quais trabalhamos.

2.3.1 Encapsulate Field

Este *refactoring* é sugerido sempre que houver um campo com acesso público na classe. Para efetuar *Encapsulate Field* basta tornar o campo privado, prover métodos *get* e *set* públicos para acessá-lo e atualizar as referências feitas ao campo para referências aos métodos. Na Figura 2.1 encontra-se representado o encapsulamento do campo `nome` da classe `Aluno`.

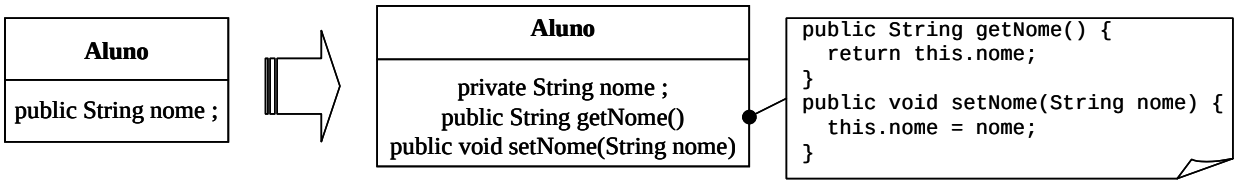

Figura 2.1: Representação do *Encapsulate Field*.

Mais detalhes sobre *Encapsulate Field* é encontrado em [Fowler, 1999], página 206.

2.3.2 Move Field

É aplicado principalmente quando um campo é, ou será utilizado por outra classe mais do que pela própria classe onde este se encontra. Para efetuar o *refactoring Move Field* deve-se criar um novo campo na classe destino, alterar todas as referências feitas ao campo e removê-lo da classe origem.

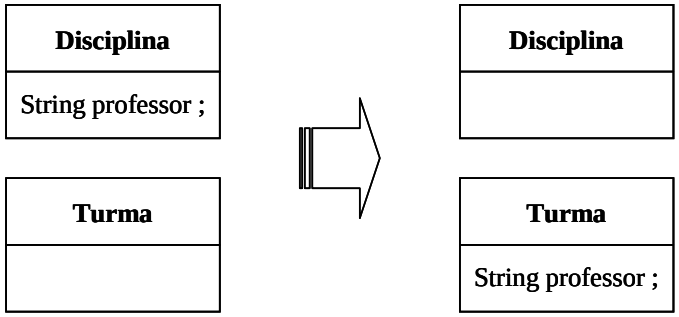

Figura 2.2: Representação do *Move Field*.

Exemplo: Sejam as classes simplificadas do Exemplo 2.1:

```
class Disciplina {
 String professor;
 Turma turma;

 public String toString() {
  return turma.codigo + " - " + professor;
 }
}

class Turma {
 String codigo;
}
```

Exemplo 2.1: *Move Field* (antes).

Deseja-se mover o campo `professor` da classe `Disciplina` para a classe `Turma`. Porém, é importante ressaltar que pode haver métodos que fazem referência a este campo, como o método `toString()` apresentado. Então, as transformações de código para alcançar este *refactoring* são:

1) cria um novo campo na classe `Turma`;

2) redireciona os métodos e;

3) remove o campo `professor` da classe `Disciplina`.

O resultado da aplicação deste *refactoring* no código do Exemplo 2.1 é apresentado no Exemplo 2.2.

```
class Disciplina {
 Turma turma;

 public String toString() {
  return turma.codigo + " - " + turma.professor;
 }
}

class Turma {
 String codigo;
 String professor;
}
```

Exemplo 2.2: *Move Field* (depois).

Mais detalhes sobre *Move Field* pode ser encontrado em [Fowler, 1999], página 146.

2.3.3 Move Method

Uma das aplicações deste *refactoring* é quando um método está utilizando, ou irá utilizar mais características exteriores do que as características da própria classe onde este se encontra definido. Ou ainda, quando outra classe utiliza este método mais do que a sua classe origem.

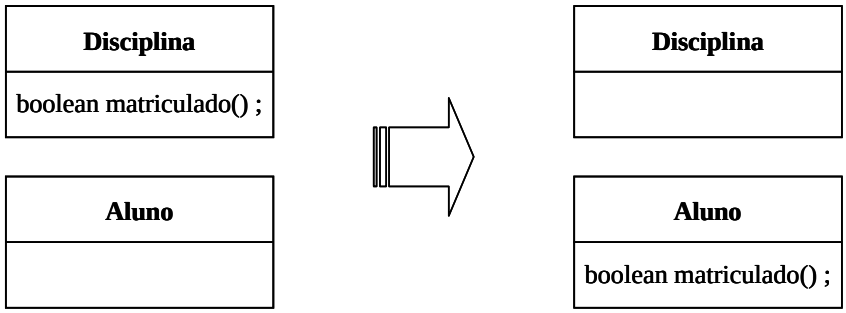

Figura 2.3: Representação do *Move Method*.

Para efetuar *Move Method*, deve-se criar um novo método na classe destino com um corpo similar ao corpo do método da classe original e transformar o método antigo em um simples redirecionador (ou removê-lo atualizando as referências).

```
class Disciplina {
 Aluno [] matriculados;
}

class Aluno {
 int mat;

 boolean matriculado(Disciplina d) {
  for(int i=0; i<d.matriculados.length; i++) {
   if (d.matriculados[i].mat == mat) return (true);
  }
  return (false);
 }
}
```

Exemplo 2.3: Classes Disciplina e Aluno antes do *Move Method*.

Um *refactoring Move Method* pode ser verificada nos exemplos de código onde o método `matriculado` da classe `Disciplina` é movido para a classe `Aluno`. O resultado do *Move Method* aplicado no código do Exemplo 2.3 é ilustrado no Exemplo 2.4.

```
class Disciplina {
 Aluno [] matriculados;

 boolean matriculado(Aluno a) {
  for(int i=0; i<matriculados.length; i++) {
   if (matriculados[i].matricula == a.matricula) return (true);
  }
  return (false);
 }
}

class Aluno {
 int matricula;
}
```

Exemplo 2.4: Classes `Disciplina` e `Aluno` alteradas.

No caso do acima, o método da classe origem foi completamente removido e atualizado as referências. Entretanto, se houver muitas referências ao método em questão, uma boa técnica é deixá-lo na classe origem como um redirecionador para o novo método.

Mais informações sobre a *refabricação Move Method* é encontrada em [Fowler, 1999], pagina 142.

2.3.4 Extract Class

A aplicação de *Extract Class* é sugerida quando se tem uma classe desempenhando um papel que deveria ser atribuído a duas. O mecanismo para efetuar este *refactoring* se baseia na criação uma nova classe, movendo os campos e métodos relevantes da classe antiga para a nova classe. Para mover os campos e métodos para a nova classe, utiliza-se os *refactorings Move Field* e *Move Method* descritos nas seções 2.3.2 e 2.3.3, respectivamente.

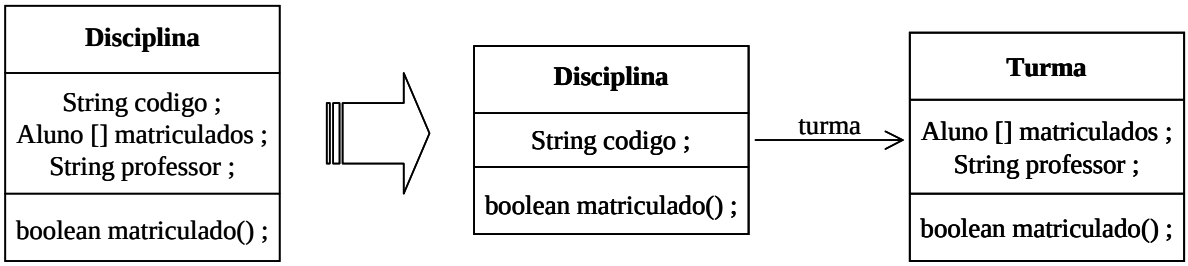

Figura 2.4: Representação de *Extract Class*.

```
class Disciplina {
 String codigo;
 String professor;
 Aluno [] matriculados;

 boolean matriculado(Aluno a) {
  for(int i=0; i<matriculados.length; i++) {
   if (matriculados[i].matricula == a.matricula) return (true);
  }
  return (false);
 }
}
```

Exemplo 2.5: *Extract Class* (antes).

O Exemplo 2.5 mostra a classe `Disciplina` onde será aplicado um *Extract Class*. Nesta classe pode-se separar as informações referentes à turma. O primeiro passo é criar uma classe vazia, que recebe o nome de `Turma`. É necessário ainda criar uma ligação na classe `Disciplina` para referenciar a nova classe. Esta pode ser criada da seguinte forma:

```
Turma turma = new Turma();
```

Define-se então, quais atributos irão para a nova classe. Neste caso, os campos `professor` e `matriculados`, e o método `matriculado` foram movidos para a nova classe.

```
class Turma {
 String professor;
 Aluno [] matriculados;

 boolean matriculado(Aluno a) {
  for(int i=0; i<matriculados.length; i++) {
   if (matriculados[i].matricula == a.matricula) return (true);
  }
  return (false);
 }
}
```

Exemplo 2.6: Classe `Turma` criada por *`Extract Class`*.

A classe `Turma` que foi criada encontra-se ilustrada no Exemplo 2.6. Os campos `professor` e `matriculados` foram movidos através do *refactoring Move Field* (seção 2.3.2) e o método `matriculado` foi movido por *Move Method* (seção 2.3.3).

```
class Disciplina {
 String codigo;
 Turma turma = new Turma();

 boolean matriculado(Aluno a) {
  return turma.matriculado(a);
 }
}
```

Exemplo 2.7: Classe `Disciplina` alterada por *`Extract Class`*.

O Exemplo 2.7 apresenta a classe `Disciplina`, após as alterações do *refactoring Extract Class*. Note que, opcionalmente, o método `matriculado` foi mantido nesta classe como um redirecionador. Como mencionado anteriormente, uma referência para a nova classe `Turma` foi adicionada na classe `Disciplina`.

Mais detalhes sobre *Extract Class* pode ser encontrado em [Fowler, 1999], página 149.

2.3.5 Inline Class

Inline Class é o reverso do *refactoring Extract Class* abordado na seção anterior, sendo sua utilização indicada quando uma classe é pequena e/ou não tem muitas responsabilidades. Este *refactoring* é conseguido movendo-se todas as características da classe para uma outra que a absorve e eliminando a classe vazia.

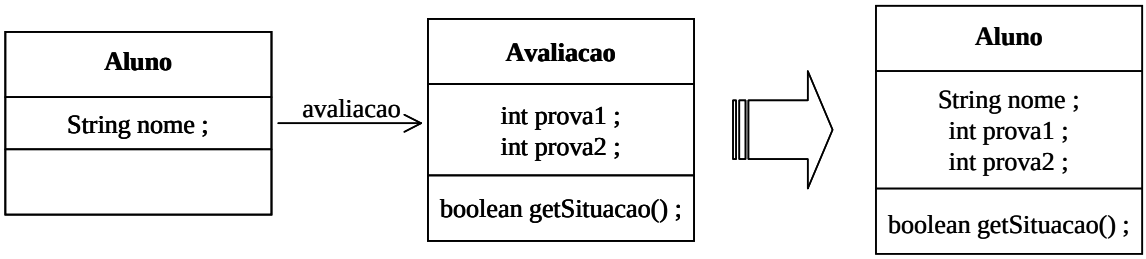

Figura 2.5: Representação de *Inline Class*.

Na representação de *Inline Class* da Figura 2.5 as classes Aluno e Avaliação se fundiram formando apenas a classe Aluno que exerce função antes desempenhada por ambas.

O *refactoring Inline Class* é mais bem detalhado em [Fowler, 1999], página 154.

3 Metaheurísicas

Técnicas de busca metaheurísticas são um conjunto de algoritmos genéricos destinados a encontrar uma boa solução, eventualmente a ótima, para um problema com vasto espaço de soluções possíveis. Estes algoritmos consistem na aplicação, em cada passo de interação, de uma heurística subordinada a qual tem que ser modelada para cada problema específico. As metaheurísticas contrariamente às heurísticas convencionais podem escapar de ótimos locais.

Metaheurísticas são comumente utilizadas para tratar problemas conhecidos na literatura como NP-difícil ou NP-completo. Para problemas desta natureza não existem algoritmos que os resolvam em tempo polinomial, sendo então enquadrados como problemas de otimização combinatória. A utilização de métodos exatos na solução é bastante restritiva, porém, na prática, geralmente é suficiente encontrar uma boa solução para o problema ao invés de uma ótima.

Exemplos de metaheurísticas são técnicas baseadas em busca local como *hill climbing*, *simulated annealing* e busca tabu (*tabu search*), e em técnicas evolutivas como algoritmos genéticos. São tratadas neste documento (seções 3.1.1, 3.1.2 e 3.1.3) apenas as metaheurísticas baseadas em busca local.

3.1 Busca Local

Dois são os aspectos chave dos problemas de otimização baseado em busca local: o primeiro é uma representação para o problema e o segundo é a escolha de uma função de otimização. A função de otimização tem como objetivo avaliar uma possível solução e comparar sua qualidade em relação a outras soluções encontradas.

As técnicas de busca local são baseadas na noção de vizinhança. Os visinhos de uma solução s são todas as soluções obtidas através de um movimento feito a partir de s. Mais especificamente, seja S o espaço de pesquisa de um problema de otimização e f a função de otimização. O conjunto $N(s) \subseteq S$ reúne um número determinado de soluções s', denominado vizinhança de s. Cada solução $s' \in N(s)$ é chamada de vizinho de s e é obtido de s a partir de uma operação chamada de movimento.

3.1.1 Hill Climbing

Hill Climbing é um procedimento de busca local que seleciona randomicamente um certo número de candidatos na vizinhança de uma determinada solução corrente. Uma vez que o melhor destes vizinhos é encontrado, sendo este é melhor que a solução corrente, ele se torna a solução corrente e o procedimento se repete. Se não existe melhor vizinho que a solução corrente, a busca termina e a solução corrente é a melhor que pode ser encontrada.

Este algoritmo de busca é chamado *Hill Climbing* porque quando o gráfico da função de avaliação possui picos indicando as melhores soluções de uma região (ótimos locais), o algoritmo se move para um ponto mais alto vizinho da solução inicial. E simplesmente vai se movimentando para o vizinho mais alto até atingir o pico do gráfico (ótimo local). A simulação se assemelha à escalada de uma montanha.

Claramente, um problema com o *Hill Climbing* é que este se prende a ótimos locais, e a melhor solução (pico) encontrada pelo algoritmo pode estar muito longe da melhor solução

global do problema. Por outro lado, o *Hill Climbing* é uma técnica simples, de fácil implementação e que tem mostrado resultados robustos para certos problemas.

Existem variações do *Hill Climbing* como por exemplo, o algoritmo pode selecionar apenas um dos vizinhos e se este for melhor que a solução corrente passa a assumir esta posição (*First Ascent Hill Climbing*). Esta implementação foi utilizada neste trabalho como é abordado na seção . Outra variação possível para o *Hill Climbing* é considerar toda a vizinhança da solução corrente, selecionando o melhor de todos os vizinhos (*Steepest Ascent Hill Climbing*).

Simulated Annealing (seção 3.1.2) e Busca Tabu (seção 3.1.3) são técnicas de busca local que também fazem pesquisa em uma estrutura de vizinhança, porém cada um deles tem uma estratégia para "saltar os vales" e escapar de ótimos locais. Ao contrário de *Hill Climbing*, os algoritmos *Simulated Annealing* e Busca Tabu são metaheurísticas por aceitarem movimentos para soluções piores do que a melhor solução encontrada.

3.1.2 Simulated Annealing

Simulated Annealing é um método de busca local proposto originalmente por Kirkpatrick [Kirkpatrick et al., 1983], e se fundamenta em uma analogia com a termodinâmica. Sua execução simula o resfriamento de um conjunto de átomos aquecidos, operação comumente utilizada na metalurgia conhecida por recozimento. O algoritmo começa sua busca a partir de uma solução inicial qualquer.

O procedimento principal consiste em um laço de repetição que gera aleatoriamente, em cada iteração, um único vizinho s' da solução corrente s. A cada geração de um vizinho s' de s, é testada a variação Δ do valor da função objetivo, isto é, $\Delta = f(s') - f(s)$. Se $\Delta < 0$, o método aceita a solução e s' passa a ser a nova solução corrente. Caso $\Delta \geq 0$ a solução vizinha candidata também poderá ser aceita, mas neste caso, com uma probabilidade $e^{-\Delta/T}$, onde T é um parâmetro do método, chamado de temperatura e que regula a probabilidade de aceitação de soluções com custo pior.

A temperatura T assume, inicialmente, um valor elevado. Após um número fixo de iterações (o qual representa o número de iterações necessárias para o sistema atingir o equilíbrio térmico em uma dada temperatura), a temperatura é gradativamente diminuída por uma razão de resfriamento α, tal que $T_n \leftarrow \alpha * T_{n-1}$, sendo $0 < \alpha < 1$. Com esse procedimento, dá-se, no início uma chance maior para escapar de ótimos locais e, à medida que T se aproxima de zero, o algoritmo comporta-se como o *Hill Climbing*, uma vez que diminui a probabilidade de se aceitar movimentos de piora ($T \rightarrow 0 \Rightarrow e^{-\Delta/T} \rightarrow 0$).

O procedimento pára quando a temperatura chega a um valor próximo de zero e nenhuma solução que piore o valor da melhor solução é mais aceita, isto é, quando o sistema está estável. A solução obtida quando o sistema encontra-se nesta situação evidencia o encontro de um ótimo local.

Os parâmetros de controle do procedimento são a razão de resfriamento α, o número de iterações para cada temperatura (`SAmax`) e a temperatura inicial T_0. A Figura 3.1 apresenta o algoritmo *Simulated Annealing* básico.

```
Procedimento S.A. (f(.), N(.), α, SAmax, T0, s)
01 s* ← s;     // Melhor solução obtida até então
02 IterT ← 0; // Número de iterações na temperatura T
03 T  ← T0;   // Temperatura corrente
04 enquanto (T > 0) faça
05  enquanto (IterT < SAmax) faça
06   IterT ← IterT + 1;
07   Gere um vizinho qualquer s' ∈ N(s);
08   Δ = f(s') – f(s);
09   se (Δ < 0) então
10     s ← s';
11     se (f(s') < f(s*)) então
12      s* ← s';
12    senão
13     Tome x ∈ [0,1];
13     se (x < e-Δ/T) então
14      s ← s';
15   fim-se;
16  fim-enquanto;
17  T ← α * T;
18  IterT ← 0;
19 fim-enquanto;
20 s ← s*;
21 Retorne s;
Fim S.A.;
```

Figura 3.1: Algoritmo *Simulated Annealing*.

Algoritmos baseados em *Simulated Annealing* geralmente incluem reaquecimento seguido de um novo processo de resfriamento, utilizado quando a quantidade de movimentos consecutivamente rejeitados é alta. É comum também trabalhar nas temperaturas mais altas com taxa de resfriamento menor e aumentá-la quando a temperatura reduzir.

3.1.3 Busca Tabu

Busca Tabu (Tabu Search) é um procedimento iterativo para resolver problemas de otimização combinatória discreta. É utilizada uma estrutura de memória para guiar um método de subida a continuar sua exploração ao espaço de possíveis soluções mesmo na ausência de movimentos de melhora. Para evitar o retorno a ótimos locais e conseqüentemente a ciclagem do algoritmo, alguns movimentos são proibidos (tabu). A lista de movimentos tabu é formada pelo inverso dos últimos movimentos feitos. Ocasionalmente, um movimento tabu pode ser aceito, isto ocorre quando o movimento atende a um critério chamado critério de aspiração.

A execução do algoritmo de Busca Tabu ocorre basicamente da seguinte forma: começando de uma solução inicial s_0, é explorado a cada iteração um subconjunto V da vizinhança N(s) da solução corrente s. O membro s' de V com menor valor segundo a função de otimização f torna-se a nova solução corrente, mesmo que s' seja pior que s. A estratégia de escolha do melhor vizinho pode fazer com que o algoritmo cicle, isto é, retorne a uma solução já gerada anteriormente.

A forma de evitar a ciclagem é proibindo movimentos que sejam o inverso dos últimos movimentos feitos pelo algoritmo. A lista tabu funciona como uma fila (FIFO) de tamanho fixo, isto é, quando um novo movimento é adicionado a lista o mais antigo sai. O tamanho da lista tabu é um dos parâmetros do algoritmo.

Se por um lado a lista tabu reduz o risco de ciclagem, por outro ela também pode proibir movimentos para soluções que ainda não foram visitadas. Assim existe uma função de aspiração, que é um mecanismo que retira, sob certas circunstâncias o status tabu de um

movimento. Um exemplo simples de aplicação desta idéia é aceitar um movimento, mesmo ele sendo tabu, se ele for melhor que a melhor solução encontrada até então.

Dois critério de parada, geralmente são utilizados para interromper a execução do algoritmo. O primeiro é número máximo de iterações sem melhora do valor da melhor solução encontrada e o segundo é quando o valor da melhor solução chega a um limite de qualidade conhecido. Esse segundo critério evita a execução desnecessária do algoritmo quando a solução ótima é encontrada ou quando um solução é julgada suficientemente boa.

```
Procedimento BT (f(.), N(.), A(.), |V|, fmin, |T|, BTmax, s)
01 s* ← s;     // Melhor solução obtida até então
02 Iter ← 0;  // Contador de iterações
03 MelhorIter ← 0;  // Iteração mais recente que forneceu s*
04 T  ← 0₀;   // Lista tabu
05 Inicializa a função de aspiração
06 enquanto (f(s) > fmin e Iter-MelhorIter<BTmax) faça
07  IterT ← IterT + 1;
08   Seja s' ← melhor elemento de V⊂N(s) tal que o movimento
       não seja tabu ou atenda a condição de aspiração.
09  T ← T – {movimento mais antigo} + {movimento gerador s'};
10  Atualiza a função de aspiração A
11  s ← s';
12  se (f(s) < f(s*)) então
13   s* ← s';
14   MelhorIter ← Iter;
15  fim-se;
16 fim-enquanto;
17 s ← s*;
18 Retorne s;
Fim BT;
```

Figura 3.2: Algoritmo Busca Tabu.

A Figura 3.2 apresenta o pseudocódigo do algoritmo básico de Busca Tabu. Neste procedimento, `fmin` é o valor mínimo conhecido da função de otimização f. O valor de `|T|` indica o tamanho da lista tabu e `BTmax` é o maior número de interações sem melhora para interromper a execução do algoritmo.

A chave para aplicação de metaheurísticas é formular o problema como um problema de busca por otimização, e definir fatores para a escolha do algoritmo. Todas as técnicas requerem uma representação para uma solução (indivíduo) e a definição de uma função de otimização. No capítulo 7 apresentamos um problema da engenharia de software que pode ser modelado como um problema de otimização.

No capítulo seguinte é apresentada uma linguagem de meta-programação utilizada para manipulação de programas Java. Esta linguagem é de importante para se obter métricas de sistemas (capítulo 5) e para aplicação de *refactoring* ao código dos programas (capítulo 6).

4 A Linguagem MetaJ

MetaJ é uma linguagem de domínio específico para meta-programação, ou seja, uma linguagem de programação cujo objeto de manipulação são programas. A linguagem utilizada nos programas manipulados por MetaJ é referenciada neste documento como linguagem objeto. Este capítulo não tem como objetivo uma abordagem ampla de MetaJ, sendo apresentados apenas os conceitos e estruturas principais para seu entendimento e utilização.

A implementação de MetaJ foi feita em Java, porém, o programa que é manipulado (programa objeto) por ela pode ter sido codificado em qualquer linguagem de programação suportada por MetaJ. O programa objeto pode ainda possuir módulos em diferentes linguagens, sendo uma situação típica, o software ter componentes descritos em Java, JSP, XML, HTML. Entretanto, neste projeto foram feitos experimentos apenas com programas de código Java puro.

A aplicação de MetaJ é bastante ampla, sendo esta definida para descrever qualquer tipo de manipulação de código como análise e detecção de padrões[], transformação de programas[] ou geração automática de código[]. Neste projeto, a linguagem MetaJ foi usada em várias etapas como a descrição de *refactorings* e a obtenção de métricas de software.

Um conceito importante de MetaJ é a linguagem de regras utilizada para descrever modelos representativos de programas (ou fragmentos de programas) da linguagem objeto. Os modelos descritos em linguagem de regras são os *templates* MetaJ abordados na seção 4.1. Uma linguagem de regras para uma linguagem objeto é uma linguagem através da qual podemos escrever *templates* que representam estruturas da linguagem objeto. A linguagem objeto está contida na linguagem de regras.

4.1 *Templates* Meta-J

Os *templates* são esquemas descritos em linguagem de regras que representam um subconjunto de programas (ou fragmentos de programas) da linguagem objeto. As linguagens de regras possuem todas as construções sintáticas da linguagem objeto e duas novas estruturas: as variáveis de manipulação e os marcadores de opcionais.

Variáveis de manipulação são estruturas que representa a ocorrência de um tipo de construção sintática da linguagem objeto nos elementos do subconjunto representado pelo *template* onde ela aparece. O tipo de uma variável de manipulação é o tipo da construção sintática representada por ela. Existem duas classes de variáveis de manipulação: *Simple* e *Set*. Que representam respectivamente, a ocorrência de uma única ou de várias construções sintáticas de um mesmo tipo.

Marcadores de opcionais delimitam construções sintáticas que podem ou não estar presentes nos elementos do subconjunto representado pelo *template* que os contém. Os marcadores de opcionais tornam o modelo mais geral.

A linguagem de regras é totalmente dependente da linguagem objeto, neste trabalho é abordado apenas a linguagem de regras de Java (Java Meta-J) a qual é utilizada na descrição dos *templates*. Em Java Meta-J a sintaxe para utilização de uma variável de manipulação é o caractere especial `#` seguido pelo tipo da variável, dois pontos (`:`) e um identificador. A sintaxe para os marcadores opcionais é `#[` construções sintáticas opcionais `]#`.

A estrutura de um *template* descrito em qualquer linguagem de regras deve seguir o modelo do Exemplo 4.1.

```
[ DeclaraçãoDePacote ]
language = <LinguagemObjeto>template [ TipoDoTemplate ]
        <Identificador> {
  [ Esquema ]
}
```

Exemplo 4.1: Modelo de um *template* MetaJ.

No modelo acima: as estruturas delimitadas por colchetes são opcionais; `<LinguagemObjeto>` é a linguagem para a qual o *template* foi definido; o `TipoDoTemplate` pode ser qualquer tipo de construção sintática da linguagem objeto sendo opcional quando este for do tipo *CompilationUnit*; o `<Identificador>` é o nome do *template*; e o `Esquema` representa o corpo do *template* descrito em linguagem de regras.

```
1  package templates.java.units;
2  language = java;
3  template JavaFile {
4    #[#PackageDeclaration:pck]#
5    #[#ImportDeclarationSet:ids]#
6    #[#TypeDeclarationSet:tds]#
7  }
```

Exemplo 4.2: *Template JavaFile*.

É apresentado no Exemplo 4.2 um *template* descrito para linguagem Java (linha 2) que representa qualquer tipo de arquivo Java válido. Note que, por não declarar o tipo do *template*, fica definido que seu tipo é *CompilationUnit*. Neste exemplo podem ser identificados três tipos de variáveis de manipulação: `PackageDeclaration` (linha 4), `ImportDeclarationSet` (linha 5) e `TypeDeclarationSet` (linha 6). A primeira é *Simple* e as outras duas variáveis são *Set*. Nas linhas 4, 5 e 6 ocorrem marcadores opcionais para todas as variáveis de manipulações do *template*.

Não são apresentados todos os tipos de variáveis de manipulação uma vez que são numerosas e dependentes da linguagem objeto. Ao invés disso, são usados exemplos de *templates* para ilustrar a utilização das variáveis com informações sobre elas. No Exemplo 4.2, a variável `PackageDeclaration` representa uma declaração de pacote do arquivo Java, `ImportDeclarationSet` representa o conjunto de importações e `TypeDeclarationSet` os tipos Java (classes e interfaces) que podem ter sido declarados no arquivo.

Qualquer fragmento de código da linguagem objeto sintaticamente correto pode fazer parte do esquema de um *template* MetaJ. Um fragmento de código dentro de um *template* indica que todos os elementos do conjunto representado pelo *template* devem conter este código. Tanto variáveis de manipulação como fragmentos de código podem aparecer como opcionais em uma regra através da utilização dos marcadores de opcionais.

```
1  language = java;
2  template #TypeDeclaration ClassDeclaration2 {
3    public class #Identifier:className #[#ClassExtends:ce]#
              #[#ImplementsClause:ic]# {
4      #[#ClassBodyDeclarationSet:cbds1]#
5      public String name;
6      #[#ClassBodyDeclarationSet:cbds2]#
7    }
8  }
```

Exemplo 4.3: *Template* de declaração de classe com campo *name*.

O *template* do Exemplo 4.3 é do tipo `TypeDeclaration` e representa toda declaração de classe pública que possua um campo `name` do tipo `String`. Note que aparece código Java (linguagem objeto) no esquema do *template*, como na linha 3 `public class` e na linha 5

`public String name;`. Novos tipos de variáveis de manipulação estão presentes neste *template*, sendo elas: `Identifier` representa um identificador de Java, neste caso, o identificador é o nome da classe; `ClassExtends` representa uma superclasse, sendo opcional para os membros deste conjunto; `ImplementsClause` também opcional, representa as interfaces implementadas; e `ClassBodyDeclarationSet` que ocorre nas linhas 4 e 6 representam declarações de membros da classe.

Todo *template* MetaJ deve ser gravado em um arquivo ".metaj". Os *templates* são mapeados para a linguagem Java pelo compilador de regras de MetaJ. Este compilador cria uma classe Java referente a cada *template*. A classe referente ao *template* possui o mesmo nome e métodos públicos para acesso às variáveis de manipulações declaradas no esquema do *template* como destacado na Figura 2.1.

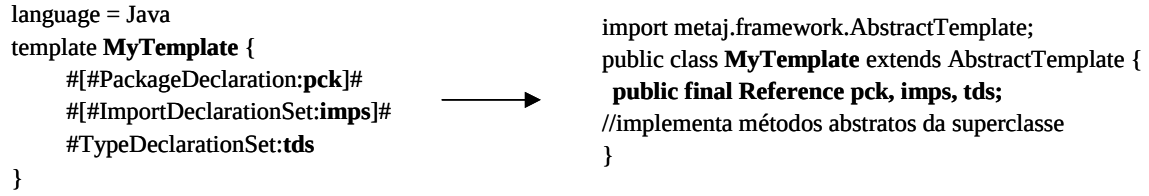

Figura 4.1: Compilação de um *template* para classe Java.

Como objetos Java, os *templates* têm vários métodos associados a eles sendo os principais métodos: *match* e *print*. O casamento (*match*) de um *template* pode ser feito com um arquivo da linguagem objeto ou com uma string e a impressão do *template* é feita em um arquivo passado como parâmetro ao método *print*.

```
1  import templates.java.units.JavaFile;
2  public class TesteJavaFile {
3    public static void main(String[] args) {
4      JavaFile tjf = new JavaFile();
5      if ( tjf.match(args[0]) )
6        System.out.println("It is a Java File")
7    }
8  }
```

Exemplo 4.4: Utilização do *template JavaFile*.

A declaração, instanciação e utilização dos *templates* são feitas como qualquer objeto Java como ilustrado no Exemplo 4.4.

4.2 Modelagem Através de Árvores

A idéia básica de MetaJ é utilizar árvores sintáticas para representar internamente os programas objeto. A linguagem define os tipos nodos da árvore sintática e oferece operações sobre estas árvores e sub-árvores. Os três tipos de nodos que podem estar presentes nas arvores sintáticas são os nodos sintáticos, nodos variáveis e nodos marcadores de trecho opcional.

```
import java.io.*;
import javax.swing.*;
class MyClass {
  . . .
}
interface MyInterface {
  . . .
}
```

Exemplo 4.5: Código apresentando dois tipos Java.

Os nodos sintáticos da árvore representam estruturas da linguagem objeto. Um programa da linguagem objeto quando transformado em árvore sintática possui apenas este tipo de nodo. Na Figura 4.2 é ilustrado a árvore sintática construída a partir do trecho de código Java do Exemplo 4.5.

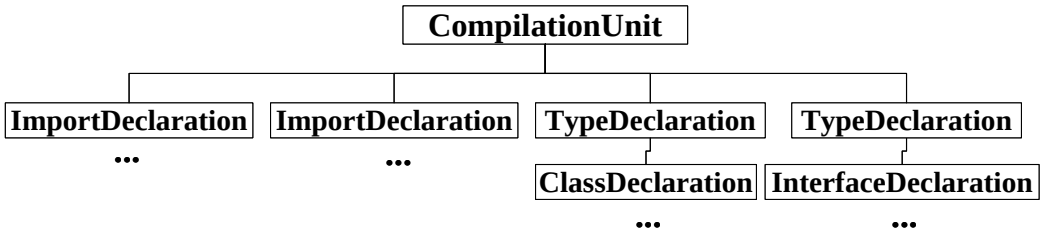

Figura 4.2: Árvore sintática de um programa Java.

Templates MetaJ também são transformados em árvores sintáticas, uma vez que eles representam programas ou fragmentos de programas na linguagem objeto. Os nodos variáveis e nodos de marcadores opcionais aparecem quando *templates* são transformados em árvores.

Os nodos variáveis representam a possibilidade de qualquer estrutura do tipo da variável, ou seja, eles definem a ocorrência de uma construção sintática de determinado tipo sem o armazenamento do seu conteúdo na árvore. Os nodos variáveis são colocados na árvore quando ocorre uma variável de manipulação no *template*.

Os nodos marcadores de trecho opcional indicam os limites das estruturas opcionais do programa. Antes da primeira estrutura opcional da árvore é colocado um nó do tipo *OptOpen* (`#[`) e após a última estrutura opcional é colocado um *OptClose* (`]#`). Estes nodos são colocados na árvore quando ocorrem marcadores de opcionais no *template*.

```
language = java;
template JavaFile {
  #[#ImportDeclarationSet:imps]#
  class MyClass {
    . . .
  }
  #TypeDeclaration:td
}
```

Exemplo 4.6: *Template CompilationUnit* com dois tipos Java.

O *template* do exemplo acima, quando transformado em árvore, apresenta os três tipos de novos definidos para MetaJ como pode ser verificado na Figura 4.3.

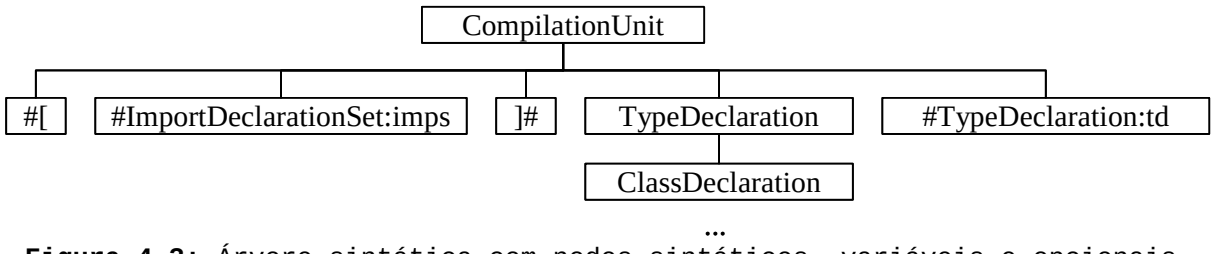

Figura 4.3: Árvore sintática com nodos sintáticos, variáveis e opcionais.

Para manipulação das árvores sintáticas foram definidas as seguintes operações sobre elas: casar, reconstruir e imprimir.

A operação de casar verifica se uma árvore está de acordo com um modelo. Este modelo pode ser um *template* ou um arquivo descrito na linguagem objeto. A operação de casar associa valores às variáveis livres da árvore e retorna um booleano indicando se o casamento foi bem sucedido.

A operação de reconstruir constrói uma nova árvore substituindo as variáveis pelo conteúdo associados a elas e removendo os marcadores de opcionais. A referência à árvore antiga é atualizada para a nova árvore.

A operação de imprimir reconstrói a árvore (pela operação reconstruir) e retorna o conteúdo da árvore como uma string.

4.3 Referência e Iterador de Árvores

Para facilitar o caminhamento dentro do código do programa objeto, foram criados os iteradores de árvores (padrão *Iterator* [Gamma et al., 1995]). Um iterador é criado a partir de uma referência a um nodo da árvore. Cada nodo possui uma referência da classe *Reference*, e esta classe oferece o método `getIterator()` que retorna o iterador.

Referências são objetos que encapsulam um trecho de programa e disponibilizam operações para manipulá-lo. A referência aponta para a raiz da sub-árvore que representa o trecho do programa.

Dentre os métodos da classe *Reference*, temos:

- *match*: compara duas referências elemento por elemento da lista;
- *print*: converte a lista de trechos em uma string resultante da concatenação da impressão de cada elemento da lista;
- *getIterator*: retorna um iterador para o trecho "apontado" pela referência;

Os iteradores de árvores oferecem cinco operações para serem aplicadas ao trecho de programa (sub-árvore). Sendo elas:

- *nextIn*: transfere o apontador do iterador para o próximo nodo da sub-árvore, sendo a navegação é feita por uma busca em profundidade. O método *nextIn* percorre todos os nodos da sub-árvore.
- *nextOver*: transfere o apontador iterador para o próximo nodo irmão (visinho à direita) da sub-árvore.
- *hasNextIn*: verifica se existem elementos para serem explorados pelo *nextIn*.
- *hasNextOver*: verifica se existem elementos para serem explorados pelo *nextOver*.
- *getRoot*: retorna uma nova referência para o nodo apontado pelo iterador.

```
1  import metaj.framework.template.environment.*;
2  import templates.java.units.JavaFile;
3  public class JavaTypes {
4    public static void main(String args[]) {
5      JavaFile file = new JavaFile();
6      if (file.match(args[0])) {
7        Reference r0 = file.rebuild();
8        Iterator it = r0.getIterator ();
9        int numberOfTypes = 0;
10       if (it.hasNextIn()) it.nextIn(); // Passa ao nodo CompilationUnit
11       if (it.hasNextIn()) it.nextIn(); // Interno ao CompilationUnit
12       while (it.hasNextOver()){
13         if (it.getRoot().getType().equals("JAVA.TypeDeclaration"))
14           numberOfTypes++;
15         it.nextOver();
16       }
17       System.out.println("Encontrado "+numberOfTypes+" tipos java.");
18     }
19   }
20 }
```

Exemplo 4.7: Aplicação do iterador.

O código Java do Exemplo 2.1 ilustra uma aplicação para o uso de iteradores. O programa conta os tipos Java (classes e interfaces) declarados em um arquivo. É necessário importar o pacote `metaj.framework.template.environment` (linha 1) para utilizar as classes `Reference` e `Iterator`. O *template* `JavaFile`, que se encontra no pacote `templates.java.units` também precisa ser importado (linha 2).

A execução deste código ocorre da seguinte forma:

1. tenta casar o *template* `JavaFile` com o arquivo Java cujo nome é passado como parâmetro para o método `main` (linha 6);
2. se o arquivo casar com o *template* é construída a árvore do arquivo e pega-se a raiz da árvore (linha 7);
3. é obtido um iterador para a árvore do arquivo (linha 8);
4. pelo uso no método `nextIn` do iterador (linha 10 e 11) atingido o nível da árvore onde se encontram das declarações de pacote, importações e tipos;
5. enquanto houver nodos vizinhos (linha 12), são feitos os itens 6 e 7;
6. se o nodo é do tipo `JAVA.TypeDeclaration` o contador de tipos é incrementado (linhas 13 e 14);
7. passa ao próximo nodo visinho pelo método `nextOver` do iterador (linha 15);

Na próxima parte deste documento é apresentada a implementação do projeto. A linguagem MetaJ foi utilizada em algumas fases do projeto para dar suporte a manipulação do código dos programas. A descrição dos *refactorings* foi feita em MetaJ, como abordado no próximo capítulo, bem como a obtenção de informações do sistema a ser manipulado e que é abordado no capítulo 5.

PARTE II: DESENVOLVIMENTO

5 Obtenção de Métricas

Apresenta-se neste capítulo JSystemInfo, uma ferramenta para coleta de métricas de software a partir do código fonte do sistema. Esta ferramenta utiliza recursos da linguagem de meta-programação MetaJ apresentada na capítulo 4. Uma abordagem sobre métricas de software é feita no capítulo 1, sendo no mesmo capítulo discutidas várias métricas que podem ser obtidas por JSystemInfo.

A Figura 5.1 ilustra a tela principal da ferramenta onde pode ser notado duas regiões para apresentação de informações do sistema avaliado. As informações do lado esquerdo da janela encontram-se organizadas em forma de árvore, hierarquizando as estruturas do sistema alvo. Por outro lado, as informações à direita da janela da ferramenta são textuais e numéricas, apresentando dados quantitativos a respeito do sistema.

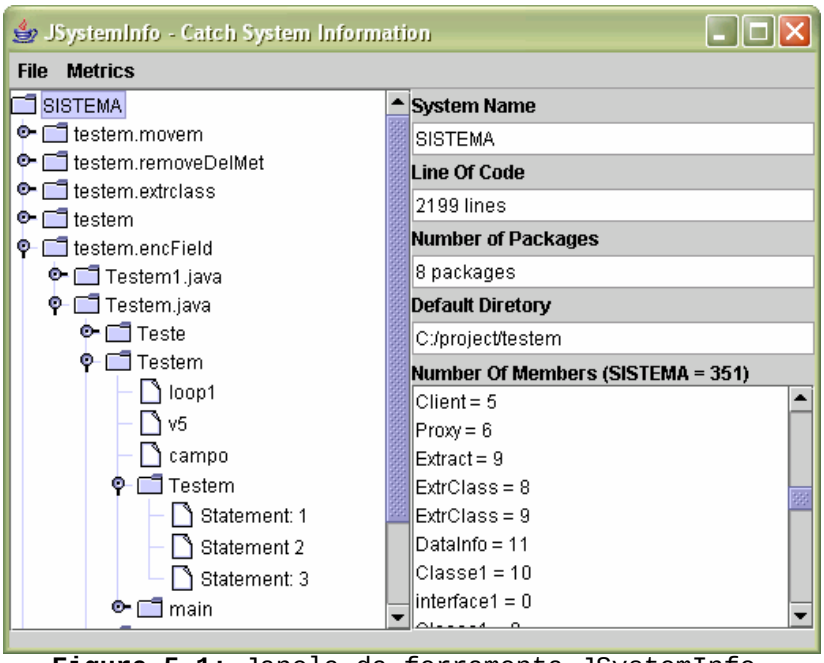

Figura 5.1: Janela da ferramenta JSystemInfo.

A hierarquia de estruturas do sistema Java avaliado é organizada da seguinte forma: o sistema possui pacotes e estes possuem arquivos Java. Internos aos arquivos encontram-se os tipos que podem ser classes ou interfaces. Cada tipo pode ter ainda campos, métodos, construtores (para classes) e outros tipos aninhados. Estruturas internas a um tipo são chamadas membros de um tipo ou simplesmente membros. Os membros que são construtores ou métodos de classe podem possuir ainda comandos. A organização destas estruturas está ilustrada na Figura 5.1 e melhor abordadas na próxima seção deste capítulo.

As informações apresentadas à direita da Janela da ferramenta são dependentes da estrutura selecionada na árvore à esquerda (Figura 5.1). Como o nodo selecionado é a raiz do sistema (`SISTEMA`), as informações apresentadas ao usuário da ferramenta são: o nome do sistema, o número total de linhas de código, o número de pacotes existentes, o diretório padrão do sistema avaliado e uma métrica de software que depende da opção selecionada no menu "`Metrics`". Na Figura 5.1, a métrica apresentada é o número de membros por tipo Java.

5.1 Organização e Coleta de Informações

Um aspecto importante para a manipulação de estruturas do sistema Java objeto é a representação destas informações. Em JSystemInfo as estruturas do sistema são organizadas em forma de árvore (Figura 5.2), sendo a raiz um nodo chamado `SystemInfo` representativo de todo o sistema. Um nível abaixo da raiz está localizado os nodos que representam os pacotes (`PackageInfo`) existentes no sistema. É obrigatória a existência de pelo menos um pacote no sistema.Cada pacote pode ter ou não arquivos Java (`JavaFileInfo`). Os arquivos Java estão no terceiro nível da árvore e dentro deles, no nível 4, encontram-se os tipos que podem ser classes (`ClassInfo`) ou interfaces (`InterfaceInfo`). As classes podem conter declarações de campos (`FieldInfo`), construtores (`ConstructorInfo`) ou métodos (`MethodInfo`) que estão no nível 5 da árvore. Construtores e métodos podem conter comandos que se classificam em: declarações de variáveis locais (`LocalVarStatement`), comandos simples (`SimpleStatement`) e comandos compostos (`ComposedStatement`). Os comandos compostos podem conter outros comandos internos a eles. Exemplos de comandos compostos em Java são: blocos `try`, laços de repetição (`do`, `while`), comandos condicionais (`if`, `switch`), entre outros.

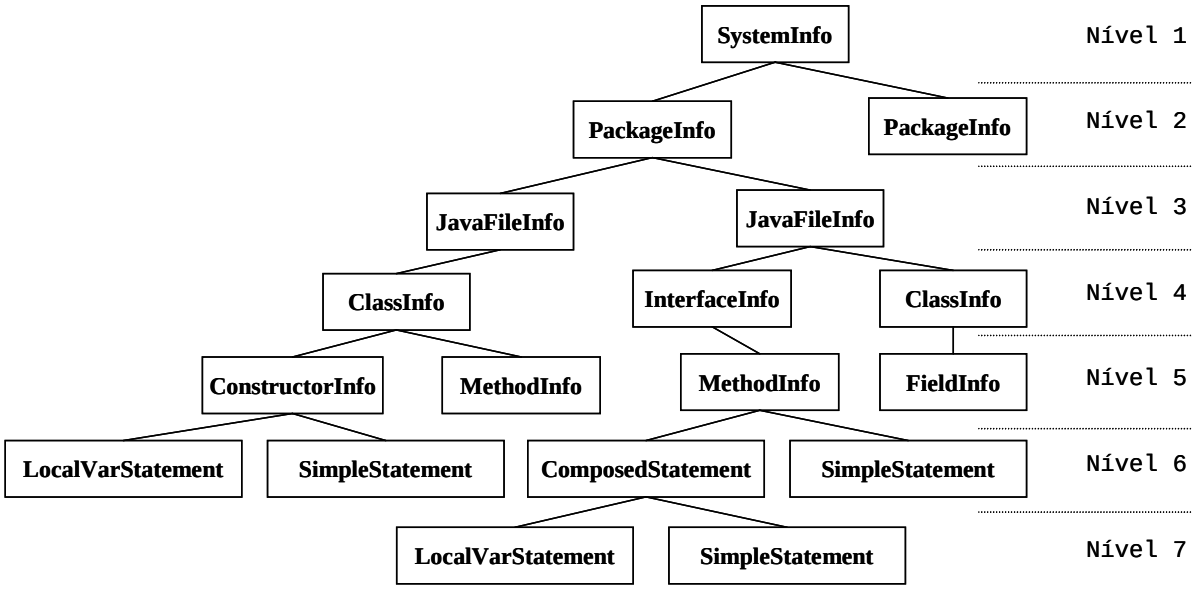

Figura 5.2: Árvore de representação de um sistema Java.

A estrutura de árvores apresentada na Figura 5.2 foi modelada pelo diagrama de classe da Figura 5.3. As classes concretas do diagrama de classes são os nodos da árvore acima. Além das classes concretas, na Figura 5.3 estão representadas também as classes abstratas e interfaces utilizadas na modelagem.

As três classes abstratas utilizadas são: `TypeInfo` abstrai os conceitos comuns de classes e interfaces, sendo uma superclasse comum para ambas; `MemberInfo` é a superclasse de qualquer membro (`FieldInfo`, `ConstructorInfo` e `MethodInfo`) de um tipo; e `ProcedureInfo`, a superclasse de `ConstructorInfo` e `MethodInfo`. Na Figura 5.3 as classes abstratas estão em fonte negrito e itálico.

Duas interfaces, `Type` e `Statement`, são utilizadas para abstrair os conceitos das estruturas Java. A interface `Type` é implementada pela classe abstrata `TypeInfo` e representa um tipo qualquer da linguagem Java. `Statement` é a interface implementa pelas classes `LocalVarStatement`, `SimpleStatement` e `ComposedStatement`. Note a existência do padrão

de projeto (*design pattern*) [Gamma et al., 1995] *Composite* na modelagem de comandos, ou seja, uma `ComposedStatement` é uma `Statement` e possui um agregado de `Statement`.

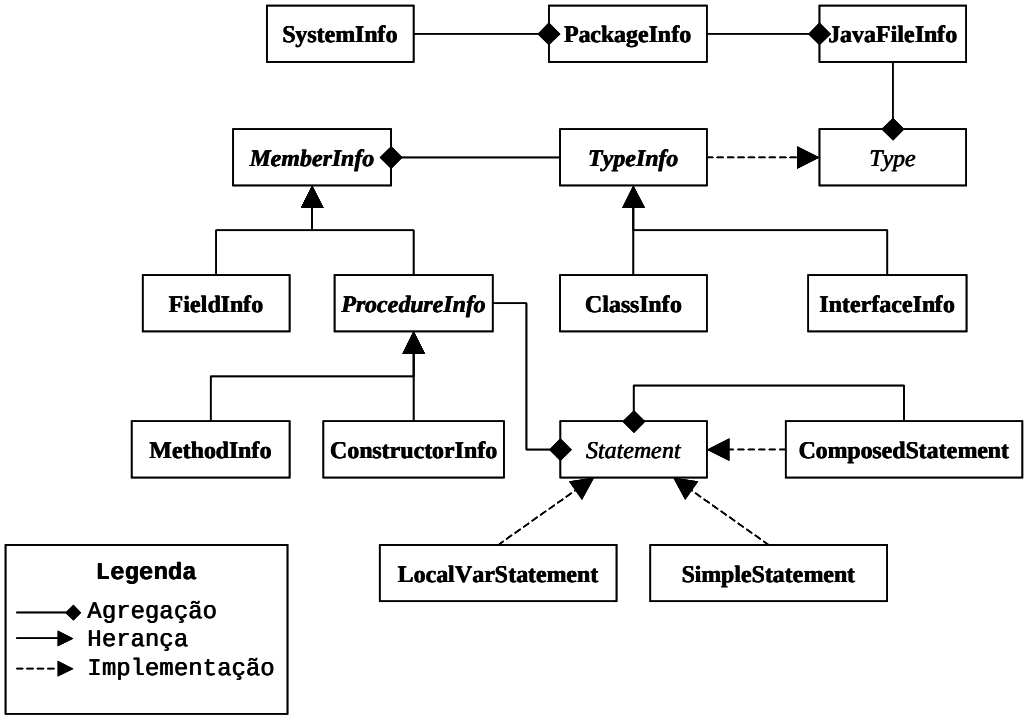

Figura 5.3: Diagrama de classes do pacote "*info*".

A obtenção das informações do sistema objeto, ou seja, o preenchimento da estrutura de dados (figuras 5.2 e 5.3) é feita por um módulo da ferramenta encontrado no pacote *collection*. Este módulo percorre todos os arquivos fonte do sistema a ser avaliado capturando as informações relevantes.

A classe principal do pacote *collection* que da inicio ao processo de coleta é uma *Thread* (pacote *java.lang*) chamada `SystemInfoCollection`. O construtor desta classe pede dois parâmetros, uma String com o nome do pacote padrão e um *File* (pacote *java.io*) que representa o diretório raiz do sistema objeto. Antes de disparar a execução de `SystemInfoCollection` é preciso criar uma instancia de `SystemInfo`, que possui instância única (padrão *Singleton* [Gamma et al., 1995]).

Para se obter todas as informações requeridas pela estrutura de dados representativa do sistema objeto, são necessárias três fases distintas. Na primeira fase a *Thread* captura informações como linhas de código, nome, e diretório do sistema (*SystemInfo*), pacotes (`PackageInfo`) e arquivos (`JavaFileInfo`). Nesta primeira fase não há utilização da linguagem de meta-programação MetaJ, visto que a mesma não suporta fluxos de execuções simultâneas (*Threads*).

Na segunda fase são coletadas todas as informações internas aos arquivos Java do sistema, ou seja, as informações de tipos (`ClassInfo` e `InterfaceInfo`), membros (`FieldInfo`, `ConstructorInfo` e `MethodInfo`) e comandos (`LocalVarStatement`, `SimpleStatement` e `ComposedStatement`). Nesta fase as *Threads* são serializadas, ou seja, passa a existir somente um fluxo de execução podendo ser utilizado MetaJ na coleta destas informações.

A terceira fase do processo de preenchimento da estrutura de dados do pacote *info* é responsável por criar referências cruzadas entre as estruturas contidas em `SystemInfo`. Por

exemplo, quando um arquivo (`JavaFileInfo`) importa um pacote pertence ao próprio sistema, é adicionado uma referência para o `PackageInfo` equivalente à importação. Da mesma forma, se é importado uma classe, o `JavaFileInfo` passa a obter uma referência para o `ClassInfo` importado.

As referências cruzadas são criadas para importações, superclasses, superinterfaces e subtipos (*children*). Após esta terceira fase, é possível navegar entre as estruturas. Por exemplo, a partir de uma classe se pode chegar a sua superclasse ou a todas as classes do sistema que a estendem (subclasses).

É apresentado a seguir um exemplo de como as informações são obtidas pela utilização de MetaJ. Suponha a coleta de informações de uma classe do sistema tais como: nome, modificadores, superclasse, interfaces implementadas e membros da classe.

Para obter tais informações de uma classe é utilizado o *template* MetaJ `ClassDeclaration` do Exemplo 5.1.

```
1  package templates.java.classes;
2  language = java;
3  template #TypeDeclaration ClassDeclaration {
4    #[ #ClassModifiers:cm ]# class #Identifier:className #[
   #ClassExtends:ce ]# #[ #ImplementsClause:ic ]#
5    #ClassBody:cb
6  }
```

Exemplo 5.1: *Template ClassDeclaration*.

O código do *template* acima descreve um esquema representativo de qualquer classe Java, pois toda classe Java possui uma assinatura composta de modificadores (opcional) seguido da palavra reservada `class`, um identificador, uma superclasse (opcional) e interfaces implementadas (opcional). Além da assinatura, toda classe Java possui um corpo que é onde se encontram declarados os membros. A compilação de um *template* por MetaJ gera uma classe Java de mesmo nome do *template* como discutido no capítulo 4.

O código Java do Exemplo 5.2 é um meta-programa pois este foi descrito para manipulação de outros programas. Para coletar as informações de uma classe Java qualquer é utiliza o *template* do Exemplo 5.1.

```
1  package metrics.collection;

2  import templates.java.classes.*;

3  public class TypeInfoCollection {
4   private String packName, typeDeclaration;
5   private TypeInfo typeInfo;

6   public void execute() {
7    ClassDeclaration tcd = new ClassDeclaration();
8    if (tcd.matchString(typeDeclaration)) {
9     String typeName = tcd.getClassName();
10    String modifiers[] = tcd.getCm().trim().split(" ");
11    String superClass = tcd.getCe();
12    String imps[] = tcd.getIc().trim().split(",");
13    typeInfo = new ClassInfo(modifiers, packName, typeName, superClass,
   imps);
14    classInfo(tcd.getCb()); // Coleta informações dos membros da classe
15   }
16  }
17 }
```

Exemplo 5.2: Meta-programa *TypeInfoCollection*.

A classe `TypeInfoCollection` pertence ao pacote *collection* da ferramenta JSystemInfo e utiliza a seguinte estratégia para obter as informações de uma classe. Tenta-se casar *template* `ClassDeclaration`, instanciado na linha 7, com uma String que contém toda a classe objeto de avaliação (linha 8). Se a classe casar com o esquema representado pelo *template*, então são capturados os dados casados com as variáveis de manipulação do *template*. As informações obtidas do código da classe são passados a uma nova instância da estrutura `ClassInfo` criada na linha 13. Informações dos membros da classe são coletadas em outro meta-programa, que é chamado mais adiante pelo método `classInfo` (linha 14).

5.2 Medidas de Tamanho, Complexidade e Acoplamento

As medições do sistema são feitas sobre a estrutura de dados do pacote *info*. Como apresentado na seção anterior, todas as informações relevantes do sistema estão organizadas na referida estrutura. O código fonte é a matéria prima para organização das informações e então se pode dizer que as medidas apresentadas nesta seção foram obtidas a partir do código fonte do sistema.

Antes de se falar das métricas propriamente ditas, vamos apresentar a forma como estas foram obtidas do sistema. Uma característica importante é que a estrutura de dados (`SystemInfo`) é estável oferecendo condições para se obter qualquer das informações. Desta forma uma opção de projeto foi utilizar o padrão *Visitor* [Gamma et al., 1995], onde as métricas são o visitante e a estrutura é o visitado.

Para que o padrão *Visitor* possa ser utilizado, as classes do pacote *info* têm que implementar um método para aceitação do visitante e os visitantes da estrutura devem ter um supertipo comum. O método de aceitação definido nas classes da estrutura é o `accept` e um supertipo de qualquer visitante é `SystemVisitor`. O pacote que contém os visitantes e outras classes utilitárias para coleta de métricas é o pacote *plugins*.

O Exemplo 5.3 apresenta de forma resumida a classe `MethodInfo` do pacote *info*. Esta classe, como todas as demais classes concretas do mesmo pacote, implementa o método `accept` (linha 5 a 7) para aceitação de um visitante do tipo `SystemVisitor`.

```
1  package metrics.info;

2  import metrics.plugins.SystemVisitor;

3  public class MethodInfo extends ProcedureInfo {
4   private String typeName;
                . . .
5   public void accept(SystemVisitor v) {
6    v.visit(this);
7   }
8  }
```

Exemplo 5.3: Classe *`MethodInfo`* do pacote *`info`*.

Todo visitante, por sua vez, deve estender direta ou indiretamente a classe abstrata `SystemVisitor` (apresentada no Exemplo 5.4) e implementar o método abstrato `getGlobalAvaliation`. Este método retorna uma avaliação (`Avaliation`) da estrutura visitada que tipicamente é uma métrica colhida do sistema objeto. Note que `SystemVisitor` possui implementações padrão dos métodos `visit` definidos para cada uma das estruturas que podem ser visitadas. Os métodos `visit` adequados para o visitante concreto devem ser redefinidos por este.

Uma subclasse de `SystemVisitor` é apresentado no Exemplo 5.5 e a classe `Avaliation` no Exemplo 5.6.

```
1  package metrics.plugins;

2  import metrics.info.*;

3  public abstract class SystemVisitor {
4   Vector avaliations = new Vector();

5   public abstract Avaliation getGlobalAvaliation();
6   public void visit(SystemInfo info) { }
7   public void visit(PackageInfo info) { }
8   public void visit(JavaFileInfo info) { }
9   public void visit(ClassInfo info) { }
10  public void visit(InterfaceInfo info) { }
11  public void visit(FieldInfo info) { }
12  public void visit(ConstructorInfo info) { }
13  public void visit(MethodInfo info) { }
14  public void visit(StatementInfo info) { }
15 }
```

Exemplo 5.4: Classe *SystemVisitor*.

```
1  package metrics.plugins.visitors;

2  import java.util.Enumeration;
3  import metrics.info.*;
4  import metrics.plugins.Avaliation;
5  import metrics.plugins.SystemVisitor;

6  public class NumberOfTypesVisitor extends SystemVisitor {
7   public Avaliation getGlobalAvaliation() {
8    Avaliation avalia = new Avaliation(SystemInfo.getInstance());
9    Avaliation[] aval = getAvaliations();
10   if (aval==null) return avalia;
11   for (int i=0; i<aval.length; i++)
12    avalia.increaseValue(aval[i].getValue());
13   return avalia;
14  }

15  public void visit(SystemInfo info) {
16   for (Enumeration packs = SystemInfo.getInstance().packages();
   packs.hasMoreElements() ; ) {
17    PackageInfo packInfo = (PackageInfo)packs.nextElement();
18    packInfo.accept(this);
19   }
20  }

21  public void visit(PackageInfo info) {
22   Avaliation packAvaliation = new Avaliation(info);
23   for (Enumeration files = info.javaFiles(); files.hasMoreElements(); ){
24    JavaFileInfo fileInfo = (JavaFileInfo)files.nextElement();
25    packAvaliation.increaseValue(fileInfo.numberOfTypes());
26   }
27   addAvaliation( packAvaliation );
28  }
29 }
```

Exemplo 5.5: Classe *NumberOfTypesVisitor*.

No Exemplo 5.5 é apresentado `NumberOfTypesVisitor`, uma subclasse de `SystemVisitor` que tem como objetivo contar o número de tipos existente em cada pacote.

Como resultado é obtido o número total de tipos Java do sistema. A classe `NumberOfTypesVisitor` é utilizada para visitar a estrutura `SystemInfo` percorrendo todos os pacotes do sistema (linhas 16 a 20). Para cada pacote (`PackageInfo`) visitado, é armazenada uma avaliação (`Avaliation`) contendo o pacote e o número de tipos do mesmo (linhas 22 a 27). Para se saber quantos tipos existem no sistema, basta invocar o método `getGlobalAvaliation`, e acessar o valor da avaliação global do sistema. A classe `Avaliation` (Exemplo 5.6) é bastante simples sendo constituída apenas de um *Object* alvo da avaliação e um valor (`value`) inteiro.

```
1  package metrics.plugins;

2  public class Avaliation {
3   private Object element;
4   private int value = 0;

5   public Avaliation(Object elem) {
6    this.element = elem;
7   }

8   public void setElement(Object element) {
9    this.element = element;
10  }

11 public void increaseValue(int number) {
12   this.value = getValue() + number;
13  }
14 }
```

Exemplo 5.6: Classe *Avaliation*.

As métricas obtidas por JSystemInfo são baseadas em métricas de software tradicionais da Engenharia de Software e encontram-se resumidamente apresentadas no capítulo 1 deste documento. Não é muito fácil diferenciar medições de tamanho, complexidade e acoplamento pois estes conceitos se relacionam. É utilizada uma divisão das métricas entre estes três grupos que não é consenso dos autores da área.

São colhidas as seguintes medidas relacionadas ao tamanho do sistema ou de componentes do sistema:

- Número de arquivos Java por pacotes (`NumberOfJavaFiles`).
- Número de tipos agrupados por pacotes (`NumberOfTypesVisitor`).
- Numero de membros existentes em cada classe (`NumberOfMembersVisitor`).
- Número de blocos de comandos (comandos compostos) por método (`NumberOfBlockStatementsVisitor`): esta medida não considera os comandos simples internos a um comando composto. Ou seja, se houver vários comandos internos a um if ou while (composto) é considerado apenas um valor na contagem.
- Numero de comandos simples em cada método (`NumberOfSimpleStatements`): conta todas os comandos de um método, inclusive os comandos internos a outros comandos.

As medidas relacionadas à complexidade são:

- Profundidade da árvore de herança por classe (`DepthOfInheritanceVisitor`).
- Número de filhos (subclasses) de cada classe (`NumberOfChildrenVisitor`).
- Número de métodos redefinidos por classe (`NumberOfOverridenMethods`).
- Número de campos públicos por classe (`NumberOfPublicFieldsVisitor`).

Finalmente, podem ser obtidas três métricas de software relacionadas ao acoplamento entre classes:

- Número de pares de tipos que possuem algum acoplamento entre elas (`NumberOfCoupleNodes`).
- Número de pares de tipos no qual um dos tipos estende ou implementa o outro (`NumberOfInheritances`).
- Número de referências para outros tipos do sistema existente em cada um dos tipos (`NumberOfReferences`).

5.3 Função de Avaliação da Qualidade

A partir das medidas propiciadas pela ferramenta JSystemInfo é possível elaborar uma função que avalie a qualidade do software baseada nas suas características. Esta função considera características desejáveis e indesejáveis do sistema. Para cada característica desejável é atribuído um valor de benefício e para cada característica indesejável, um valor de penalidade.

$$F(x) = \Sigma\ (\text{Penalidades}) - \Sigma\ (\text{Benefícios})$$

Figura 5.4: Função de avaliação da qualidade do sistema.

O valor final da função é resultado da soma de todas as penalidades, subtraído pelos benefícios atribuídos ao sistema, conforme ilustrado na Figura 5.4. As características do sistema possuem valores de penalidades/benefícios diferentes e estes valores são definidos por um arquivo de propriedade do sistema. Este arquivo de nome *FaultValue.txt* se encontra no pacote *guru.config* e os valores padrão atribuído a cada característica são apresentados na .

Característica	Valor atribuído
Penalidade por linha de código que ultrapassar o mínimo desejado	1
Penalidade por cada arquivo existente no sistema	0
Penalidade por cada tipo (classe ou interface)	0
Penalidade por cada classe	1
Penalidade por classe com menos de 10 membros	5
Penalidade por classe com mais de 20 membros	20
Penalidade por classe com mais de 30 membros	50
Penalidade por interface com mais de 30 membros	20
Penalidade por cada campo público existente no sistema	20
Penalidade por método com mais de 20 comandos simples	10
Penalidade por método com mais de 50 comandos simples	30
Penalidade por classe com árvore de hierarquia maior que 5 níveis	30
Penalidade por classe com árvore de hierarquia maior que 8 níveis	50
Penalidade por interface com arvore de hierarquia maior que 5 níveis	30
Penalidade por interface com arvore de hierarquia maior que 8 níveis	50

Penalidade por classe com mais de 6 filhos diretos	10
Penalidade por classe com mais de 10 filhos diretos	20
Penalidade por método com mais de 6 ciclos de McCabe	20
Penalidade por método com mais de 10 ciclos de McCabe	50
Prêmio por cada aresta de herança do sistema (*extends* ou *implements*)	-10 *
Penalidade por par de tipos com mais de 3 arestas de heranca/campo	10
Penalidade por par de tipos com mais de 5 arestas de referências	10
Penalidade por par de tipos com mais de 8 arestas de referências	30
Penalidade por método redefinido na subclasse	10
Premio por método definido na superclasse que não é redefinido	-20 *

* Valores negativos representam prêmios para o sistema.

Tabela 5.1: Valores das penalidades e benefícios.

Da forma como é definida a função de avaliação, seu valor seria inversamente proporcional à qualidade do sistema, ou seja, quanto menor o valor da função de avaliação, melhor é o sistema segundo a função de avaliação. É importante lembrar que apenas uma fração das características do software é medida por esta função sendo ingenuidade afirmar a respeito da qualidade do sistema baseado apenas em seu valor.

6 Programação de *Refactoring*

Este capítulo se dedica a *refactorings* descritos na linguagem de manipulação de programas JMJ [Oliveira, 2002]. JMJ é uma versão de MetaJ (capítulo 4) específica para manipulação de programas Java. São abordadas os cinco *refactorings* do capítulo 2, sendo apresentado o código implementado em JMJ para efetuar o *refactoring Move Field*. Para os demais *refactorings* não serão apresentados os códigos, porém, estes se encontram disponíveis nos anexos.

Todas os *refactorings* do capítulo são apresentadas através da descrição de sua execução, isto é, os passos necessários para se obter o *refactoring*. As manipulações feitas no código do sistema para atingir um *refactoring* são de três tipos: alteração, verificação ou geração de código. Além da execução, apresenta-se para cada *refactoring* os atributos que devem ser passados para os meta-programas e as condições impostas ao sistema objeto para permitir que o *refactoring* seja executado.

6.1 Encapsulate Field

A implementação do *Encapsulate Field* descrito nesta seção se baseia no modelo proposto por [Fowler, 1999] página 206. O objetivo deste *refactoring* é tornar um campo privado provendo métodos *get* e *set* para acessa-lo. Informações sobre este *refactoring* são introduzidas na seção 2.3.1 deste documento.

A implementação deste *refactoring* utilizado JMJ é encontrada no Anexo A e as etapas de execução são basicamente as seguintes. É verificado se não existem métodos de nome `get<campo>` e `set<campo>` na classe. Não sendo estes métodos encontrados, eles são criados. Um dos modificadores do campo a ser encapsulado passa a ser `private`, modificadores do tipo `public` ou `protected` são excluídos. Finalmente são procuradas todas as referências ao campo nas classes existentes na lista diretórios `dirClients` e estas referências são atualizadas para referências aos métodos *get* e *set* do campo. A lista `dirClients` é um vetor (classe Vector) que conta os diretórios onde pode haver possíveis classes clientes do campo.

Os campos que devem ser configurados antes da execução da *Encapsulate Field* são:

- Arquivo de entrada (`inputFile`): arquivo original do sistema que contém a classe com o campo a ser encapsulado.
- Arquivo de saída (`outputFile`): arquivo gravado pelo *refactoring* com o campo encapsulado.
- Nome da classe (`className`): nome da classe que possui o campo.
- Nome do campo (`fieldName`).

6.2 Move Field

A implementação do *Move Field* descrito nesta seção se baseia no modelo proposto por [Fowler, 1999] página 146. O objetivo deste *refactoring* é a retirada de um campo de uma classe, adicionando-o a outra. Outros conceitos de *Move Field* estão presentes na seção 2.3.2.

```
1  package jmj.refactorings.fowler.moveField;

2  import jmj.refactorings.fowler.encapsulateField.*;
3  import jmj.refactorings.fowler.util.remove.fieldDeclaration.*;
4  import jmj.refactorings.fowler.util.add.encapsulatedField.*;
```

```
5  public composed metaCode MoveField {

6   String inputSourceFile;  // Arquivo com a classe origem
7   String outputSourceFile; // Arquivo com a classe origem reestruturada
8   String inpuTargetFile;  // Arquivo com a classe destino
9   String outputTargetFile; // Arquivo com a classe destino reestruturado
10  String sourceClass;  // Nome da classe origem
11  String targetClass;  // Nome da classe destino
12  String fieldName;  // Nome do campo a mover

13  public MoveField(String isf, String osf, String itf, String otf,
              String sc, String tc, String fn) {
14   inputSourceFile = isf;
15   outputSourceFile = osf;
16   inpuTargetFile = itf;
17   outputTargetFile = otf;
18   sourceClass = sc;
19   targetClass = tc;
20   fieldName = fn;
21  }
22  public void execute() throws ActionException {
        // VerifyField
23   MetaCode rlVerifyField = #new ClassWithField(inputSourceFile);
24   rlVerifyField.setVariable ("className", getIdentifier(sourceClass));
25   rlVerifyField.setVariable("fieldName", getIdentifier(fieldName));
26   if (rlVerifyField.executeMetaCode () == SUCCESS) {
        // EncapsulateField
27    EncapsulateField mcEncapsulate = new EncapsulateField
   (inputSourceFile, outputSourceFile, sourceClass, fieldName);
        // RedirectMethod
28    RedirectMethod mcRedirectMethod;
29    try {
30     mcEncapsulate.execute();
31     mcRedirectMethod = new RedirectMethod(outputSourceFile,
   outputSourceFile, sourceClass, mcEncapsulate.getSetMethod(),
   targetClass);
32    }
33    catch (Exception e) { // O campo ja estava encapsulado
34     mcRedirectMethod = new RedirectMethod(inputSourceFile,
   outputSourceFile, sourceClass, mcEncapsulate.getSetMethod(),
   targetClass);
35    }
        // Atualiza corpo do metodo set
36    try {
37     mcRedirectMethod.execute();
38    }
39    catch (Exception e) { }
40    mcRedirectMethod.setInputFile(outputSourceFile);
41    mcRedirectMethod.setMethodName( mcEncapsulate.getGetMethod() );
        // Atualiza corpo do metodo get
42    try {
43     mcRedirectMethod.execute();
44    }
45    catch (Exception e) { }
        // RemoveField
46    RemoveField mcRemoveField = new RemoveField(outputSourceFile,
   outputSourceFile, sourceClass, fieldName);
47    mcRemoveField.execute();
        // AddEncapsulatedField
48    AddEncapsulatedField mcAddEncapsField = new AddEncapsulatedField
   (inpuTargetFile, outputTargetFile, targetClass,
```

```
   rlVerifyField.#fm.toString(), rlVerifyField.#type.toString(),
   rlVerifyField.#fieldName.toString(), rlVerifyField.#vi.toString());
49    mcAddEncapsField.execute();
         // VerifyField
50    MetaCode rlVerifySrcObj = #new ClassWithField(outputTargetFile);
51    rlVerifySrcObj.setVariable("className", getIdentifier(targetClass));
52    rlVerifySrcObj.setVariable("type", getType(sourceClass));
53    if (rlVerifySrcObj.executeMetaCode() == SUCCESS) {
         // RedirectClass
54     RedirectClass rlRedirectClass = new RedirectClass(outputTargetFile,
   outputTargetFile, targetClass, sourceClass,
   mcEncapsulate.getSetMethod(), mcEncapsulate.getGetMethod());
55     rlRedirectClass.execute();
56    }
57   }
58   else throw new ActionException("MoveField: Condition FAIL");
59  }
60 }
```

Exemplo 6.1: Refabricação *Move Field*.

É necessário que meta-programas menores sejam agrupados para se conseguir atender a todos os requisitos de um *refactoring*. O Exemplo 6.1 apresenta o código do meta-programa principal onde pode ser verificado a declaração e instanciação de outros meta- programas (linhas 23, 27, 28, 31, 34, 46, 48, 50 e 54). A execução deste *refactoring* consiste na seqüência de verificações e transformações descrita a seguir.

Primeiro é verificado a existência do campo a ser movido na classe origem. Esta tarefa é desempenhada por um meta-programa chamado `ClassWithField` (linhas 23 a 26). Se não houver o campo na classe é lançada uma exceção (linha 58). Por outro lado, se o campo for encontrado na classe, este é encapsulado (linhas 27 e 30). Encapsular um campo quer dizer tornar este campo privado e alterar todos os acessos diretos a ele em acessos via método *get* ou *set*. O encapsulamento de campo é feita pelo *refactoring Encapsulate Field* (seção 6.1).

Após o *Encapsulate Field* criar métodos *get* e *set*, o corpo de cada um destes métodos é alterado para acessar o campo em uma outra classe, ou seja, os métodos passam a acessar o campo na classe destino. Esta adaptação é feita por `RedirectMethod` nas linhas 31 a 39. Um detalhe interessante deste meta-programa é que ele cria uma referência para a classe destino, caso não seja encontrada nenhuma na classe origem.

O próximo passo é remover o campo da classe origem (linhas 46 e 47). Note que, apenas a declaração do campo é removida e não os métodos *get* e *set*. Em seguida o campo é adicionado, juntamente com outros métodos *get* e *set* na classe destino pelo meta-programa `AddEncapsulatedField` (linhas 48 e 49).

Para finalizar o *Move Field*, é verificado se existe alguma referência à classe origem na classe destino (linhas 50 a 53). Havendo referências a classe origem, são atualizados todos os acessos ao campo feitos pela classe destino. Ou seja, os acessos passam a serem feitos pelos métodos *get* e *set*, criados na própria classe destino. A atualização destas referências é feita pelo meta-programa `RedirectClass` nas linhas 54 e 55.

Para que o *refactoring Move Field* seja executado, é necessário que os seguintes campos sejam configurados:

- Arquivo de entrada da classe origem (`inputSourceFile`): este é o arquivo que possui classe é onde se encontra o campo antes do *refactoring*.

- Arquivo de saída da classe origem (`outputSourceFile`): este arquivo será criado pelo *refactoring* e irá ser um espelho do arquivo `inputSourceFile`. Exceto pela classe origem que perde o campo movido.
- Arquivo de entrada da classe destino (`inputTargetFile`): este arquivo possui a classe destino do campo, antes do *refactoring*.
- Arquivo de saída da classe destino (`outputTargetFile`): este arquivo será criada pelo *refactoring* para armazenar o arquivo `inputTargetFile` reestruturado.
- Nome da classe origem (`sourceClass`).
- Nome da classe destino (`targetClass`).
- Nome do campo a ser movido (`fieldName`).

Além dos campos acima, algumas condições devem ser satisfeitas para que o *refactoring* seja executado com sucesso. Primeiro, não deve haver um campo de mesmo nome que `fieldName` na classe destino. Bem como, não haver métodos de nome *get<fieldName>* e *set<fieldName>* na classe destino. Se houver métodos de nome *get<fieldName>* e *set<fieldName>* na classe origem, estes devem ser usados para acessar o campo em questão (sem nenhum tipo de processamento).

6.3 Move Method

Neste *refactoring*, o objetivo é transferir um método de uma classe para outra. Sua implementação pode ser encontrada no Anexo B e é baseada na proposta de *Move Method* elaborada por [Fowler, 1999] página 142. Informações sobre a utilização e um exemplo dos efeitos da aplicação deste *refactoring* ao código Java é apresentado na seção 2.3.3.

A solução proposta utiliza a linguagem de meta-programação JMJ para efetuar sucessivas verificações e transformações de código. O primeiro passo deste processo, é verificar a existência do método a ser movido na classe origem com o auxílio de um meta-programa auxiliar. Caso não seja encontrado o método na classe origem, uma exceção é lançada.

Ao ser encontrado na classe origem, é utilizado o meta-programa *AddMethod* para adicionar o método na classe destino. O corpo deste método é copiado de forma idêntica ao da classe origem. Esta operação só pode ser realizada se não houver um método de mesmo nome na classe destino.

Após adicionar o método à classe destino, são necessárias duas transformações no corpo do método para adaptá-lo a esta classe. A primeira consiste em substituir todos os atributos de acesso direto, por acesso via uma referência à classe origem existente na classe destino. Esta transformação não é feita em variáveis que são passadas ao método por parâmetro. Se não existir nenhuma referência à classe origem na classe destino, é criada uma.

A segunda transformação ocorrida no corpo do método é o oposto da primeira. Ou seja, os atributos que eram acessados indiretamente, através de uma referência à classe destino existente na classe origem, passam a serem acessados diretamente na classe destino. Ambas as transformações são efetuadas pelo meta-programa programa chamado *RedoMethod*.

Até este momento, o método teria sido movido para a classe destino, mas nenhuma referência ao método foi atualizada. Então, o meta-programa *RedirectTarget* procura acessos indiretos ao método no corpo da classe destino e substitui por acessos direto.

Finalmente, o corpo do método na classe origem pode ser removido. Entretanto, a assinatura do método é mantida e seu corpo passa a consistir apenas de um redirecionador (forward) ao mesmo método na classe destino. Esta técnica é utilizada para evitar que seja

necessário procurar todas as referências ao método em outras classes do sistema. O meta-programa *RedirectSource* é o responsável por substituir o corpo do método da classe origem por um redirecionador .

Para ser executado o *Move Method* é necessário configurar os seguintes campos:

- Arquivo de entrada da classe origem (`inputSourceFile`): este é o arquivo origem, ou seja, o arquivo que possui a classe onde o método se encontra definido antes do *refactoring*.
- Arquivo de saída da classe origem (`outputSourceFile`): este arquivo será criado pelo *refactoring* e irá ser um espelho do arquivo da classe origem. Exceto pelo método movido, que permanecerá apenas como redirecionador na classe origem.
- Arquivo de entrada da classe destino (`inputTargetFile`): este arquivo possui a classe destino do método.
- Arquivo de saída da classe destino (`outputTargetFile`): este arquivo será criado pelo *refactoring* para o arquivo da classe destino depois de reestruturado.
- Nome da classe origem (`sourceClass`)
- Nome da classe destino (`targetClass`)
- Nome do método a ser movido (`methodName`)

Algumas condições precisam ser satisfeitas pelos arquivos Java para que o *refactoring* possa ser aplicado. A classe destino do método, por exemplo, não pode ter um método de mesmo nome, mesmo recebendo parâmetros diferentes. Outra condição imposta, é que os atributos da classe origem, que são acessados pelo método, precisam ter visibilidade para a classe destino do método.

6.4 Extract Class

Este *refactoring* tem por objetivo dividir uma classe em duas, distribuindo os atributos entre as classes. O *refactoring Extract Class* utiliza *Move Field* e *Move Method* para mover os atributos da classe origem para a nova classe. Sua implementação se baseia em [Fowler, 1999], página 149, e a codificação JMJ está presente no Anexo C deste documento. Mais informações sobre o *Extract Class* são encontradas na seção 2.3.4.

Os passos que devem ser seguidos para se conseguir efetuar este *refactoring* são basicamente os quatro a seguir.

Primeiro, é preciso criar uma nova classe vazia no arquivo de saída, sendo que, esta classe pode ser criada em um arquivo Java existente, ou em um novo arquivo Java. O meta-programa auxiliar responsável por esta tarefa é chamado *CreateClass*.

Após a nova classe ter sido criada, é preciso criar uma referência na classe origem para ligá-la à nova classe. Através desta referência, poderão ser acessados os campos e métodos que serão movidos para a nova classe. Este segundo passo na execução do *refactoring* é de responsabilidade do meta-programa *AddField*.

Finalmente, o terceiro e o quarto passo consiste em mover os campos e os métodos para a nova classe. Os campos são movidos pelo *refactoring Move Field* apresentado na seção 6.2 e os métodos pelo *Move Method* apresentado na seção 6.3.

Os campos que devem ser configurados antes da execução da *Extract Class* são os seguintes:

- Arquivo de entrada da classe origem (`inputSourceFile`): este é o arquivo original que possui a classe que será dividida.

- Arquivo de saída da classe origem (`outputSourceFile`): este arquivo será criado para armazenar o arquivo origem após o *refactoring*.
- Arquivo de entrada da classe destino (`inputTargetFile`): este é o arquivo onde será criada a nova classe. Não é obrigatória a existência deste arquivo.
- Arquivo de saída da classe destino (`outputTargetFile`): este arquivo será criado para armazenar o arquivo destino depois de reestruturado. Se o arquivo destino não existir, após o *refactoring*, este arquivo conterá apenas a nova classe criada.
- Nome da classe origem (`sourceClass`)
- Nome da classe destino a ser criada (`targetClass`)
- Nome(s) do(s) campo(s) a ser(em) movido(s) para nova classe (`fields`)
- Nome(s) do(s) método(s) a ser(em) movido(s) para nova classe (`methods`)

Os campos `fields` e `methods` do *refactoring Extract Class*, são do tipo *Vector*. Estes possuem dois métodos associados que podem ser usados para adicionar um novo valor ao vetor: `addField(String field)` e `addMethod(String method)`.

Além das condições impostas pelas refabricações *Move Field* e *Move Method*, para se executar a *Extract Class* é necessário que não haja nenhuma classe no arquivo destino de mesmo nome da classe a ser criada.

6.5 Inline Class

O *refactoring Inline Class* faz o merge de duas classes, ou seja, move todas as características de uma classe para outra eliminando a primeira. Informações conceituais deste *refactoring* são apresentadas no seção 2.3.5 e agora é mostrada uma possível implementação utilizando a linguagem de manipulação de programas JMJ. Esta implementação é baseada em [Fowler, 1999], página 154.

Em *Inline Class*, como nos demais *refactorings* implementados, é necessário agrupar pequenas verificações transformações para se obter o seu resultado esperado. Estes pequenos itens de meta-programação são descritos a seguir.

A primeira transformação é feita no arquivo absorvido. Se houver uma variável de instância que seja uma referência a classe absorvedora, ela é removida. Fazendo uma analogia com *Move Method* (ou *Move Field*), a classe absorvida é a classe origem e a classe absorvedora é a classe destino da característica. A remoção da referência à classe absorvedora se justifica para evitar que esta seja movida, lembre-se que todas as características da classe serão movidas.

Os membros da classe absorvida (origem) são movidos para a classe absorvedora (destino) utilizando os *refactorings Move Field* e *Move Method*. Para que isso seja possível são feitas duas listas, uma contendo os campos e outra os métodos da classe absorvida. Após as características terem sido movidas, se houver uma referência à classe absorvida na classe absorvedora esta é também removida.

É necessário agora, atualizar todas as referências do sistema. As referências à classe origem são redirecionadas para a classe absorvedora e finalmente a classe absorvida pode ser removida. Se o arquivo que possuía a classe removida não tem mais nenhuma classe, este arquivo também é ser removido. O arquivo de *bytecode* ".class" referente à classe absorvida é removido uma vez que este não será mais necessário.

Os campos do *Inline Class* que precisam ser configurados antes de executar o *refactoring* são:

- Arquivo de entrada da classe origem (`inputSourceFile`): este é o arquivo original que possui a classe que será absorvida.
- Arquivo de saída da classe origem (`outputSourceFile`): este arquivo é gravado pelo *refactoring* contendo as demais classes existentes no mesmo arquivo da classe absorvida. Este arquivo pode não ser gravado se o arquivo de entrada da classe origem possuir apenas a classe absorvida.
- Arquivo de entrada da classe destino (`inputTargetFile`): este é o arquivo onde se encontra a classe absorvedora antes do *refactoring*.
- Arquivo de saída da classe destino (`outputTargetFile`): este arquivo é criado para armazenar a classe absorvedora depois de receber as novas características.

As condições para que este *refactoring* possa ser aplicado são as mesmas impostas por *Move Field* e *Move Method*.

7 Identificação Automática de *Refactoring*

Este capítulo argumenta sobre a aplicação de *refactoring* de forma automática, ou seja, sem nenhuma interferência do desenvolvedor na escolha dos itens de *refactoring* e nem na configuração destes. Os *refactorings* são selecionados e aplicados segundo características do sistema objeto. Estas características são avaliadas pela ferramenta JSystemInfo descrita no capítulo 5 e as implementações utilizadas dos *refactorings* são apresentadas no capítulo 6.

7.1 Relacionando Métricas a *Refactorings*

A aplicação de *refactorings* em um sistema muda muita das características do código deste sistema. Comumente os *refactorings* são aplicados de forma programada, ou seja, o desenvolvedor precisa alterar o sistema para atender um novo requisito e utiliza uma ferramenta para auxiliá-lo neste processo.

Outras vezes são utilizados os *refactorings* apenas para melhorar a qualidade do código fontes, seja para torná-lo mais legível ou para adaptá-lo a um padrão de projeto [Gamma et al., 1995]. Nestas últimas situações, apesar de geralmente serem feitas de forma programada, é possível que a aplicação de *refactoring* seja automatizada.

Os *refactorings* alteram o código fonte, influindo também na qualidade interna do software. Pode-se afirmar, por exemplo, que o *Encapsulate Field* aumenta o encapsulamento do sistema pois este reduz o número de campos públicos. Por outro lado, ao ser aplicado este *refactoring*, o número de métodos na classe do campo encapsulado é aumentado em dois.

Os *refactorings Move Field* e *Move Method* redistribuem as características entre as classes do sistema. Por outro lado, é provável que seja aumentado o acoplamento entre as classes após a aplicação destes *refactorings*. Outra característica do sistema influenciada por estes *refactorings* que movem características é a coesão das classes envolvidas.

O *Extract Class* e o *Inline class* são os *refactorings* com os quais trabalhamos que mais influenciam as características do sistema. O primeiro reduz o tamanho da classe que sofreu extração (retirada) de suas características, porém, uma nova classe é criada no sistema para receber estas características. Já o segundo aumenta a granulanidade de uma classe e reduz o número de classes do sistema, uma vez que agrupa as características de duas classes em apenas uma. Note que aumentar as responsabilidades de uma classe pode torná-la bastante complexa.

Apesar das funcionalidades semânticas do sistema não se alterarem com a aplicação de *refactorings*, a qualidade do sistema é alterado. Isto é comprovada com a avaliação da qualidade do código fonte de um sistema antes e após a aplicação de um *refactoring* e é notado que os valores encontrados dificilmente serão os mesmos. Assim se justifica a aplicação de *refactorings* com o intuído de aumentar a qualidade do código fonte de um sistema.

7.2 Heurística para Seleção e Aplicação

Não podemos afirmar que um *refactoring* sempre irá aumentar a qualidade do sistema, desta forma, é preciso prover um mecanismo que desfaça a alterações feitas por um *refactoring* caso a qualidade da versão do software gerado tenha qualidade inferior. A solução escolhida para desfazer um *refactoring* é fazer a cópia dos arquivos alterados pelo mesmo. Este controle das versões dos arquivos do sistema é feito pela classe que aplica os

refactorings (`Refactor`) cujo código é apresentado no Anexo E. A classe `Refactor` utiliza uma pilha de estados (`State`) que representam as versões do sistema. Ao ser aplicado um *refactoring*, um novo estado é adicionado a pilha e para se desfazer as alterações feitas (voltando à versão anterior do sistema), basta desempilhar o último estado.

A classe `State` grava a cópia dos arquivos alterados na pasta "...\guru\undo\files" sendo a cópia do arquivo gravado com o nome `<fileName>@State<stateNumber>`, onde `<fileName>` é o nome do arquivo original e `<stateNumber>` é o numero do estado responsável por recuperar o arquivo. Além de gravar a cópia dos arquivos, um estado (classe `State`) diz o tipo de manipulação feita no arquivo original. Dois são os tipos de manipulações possíveis: alterar/apagar o arquivo e criar um novo arquivo.

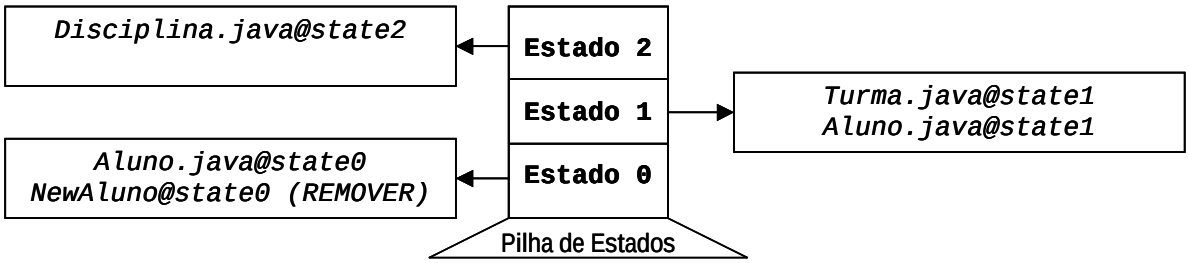

Figura 7.1: Pilha de estados (versões) do sistema.

Se o arquivo original do sistema foi alterado ou apagado, o estado grava uma cópia deste arquivo para uma possível recuperação. Se, por outro lado, um *refactoring* (como o *Extract Class*) criar um novo arquivo no sistema, o estado guarda informações de qual arquivo foi criado e uma observação "REMOVER" para uma possível futura remoção. A representação de uma pilha de estados é ilustrada na Figura 7.1.

O processo de seleção e aplicação dos *refactorings* adequados é controlado por uma heurística tradicional, conhecida como *Hill Climbing* e que é discutida na seção 3.1.1. Esta heurística de busca local aceita apenas movimentos de melhora, ou seja, para o problema em questão são aceitos os *refactorings* que aumentem a qualidade do sistema.

A implementação de *Hill Climbing* para o problema de seleção e aplicação de recfactorings é apresentado no Exemplo 7.1.

```
1  package guru.heuristic;

2  import metrics.StartMetrics;
3  import metrics.avaliation.GlobalFunction;
4  import guru.heuristic.refactRandom.*;

5  public class HillClimbing extends RefactoringRandom {
6   private int interation = 5;
7   private int refactorings = 5;
8   ClientSystem system = new ClientSystem();
9   Refactor refactor = new Refactor();

10  public void execute() {
11   int lastAvaliation = getAvaliation();
12   for (int i=0; i<interation; i++) {
13    boolean isRefactor = doRefactor();
14    if (isRefactor) {
15     int avaliation = getAvaliation();
16     if ( lastAvaliation > avaliation ) lastAvaliation = avaliation;
17     else refactor.undo();
18    }
19   }
```

```
20  }
              // Coleta informacoes do sistema
21  int getAvaliation() {
22   StartMetrics metrics = new StartMetrics(system.getPackage(),
   system.getDiretory(), system.getSystemName());
23   metrics.waitCollection();
              // Pega valor da funcao de avaliacao
24   GlobalFunction function = new GlobalFunction();
25   function.execute();
26   int avaliation = function.getSystemAvaliation().getValue();
27   return avaliation;
28  }
              // Aplicador de refactoring randomico
29  boolean doRefactor() {
30   int switchRefact = randomize(refactorings);
31   switch (switchRefact) {
32    case 0: return doEncapsulateField();
33    case 1: return doMoveField();
34    case 2: return doMoveMethod();
35    case 3: return doExtractClass();
36    case 4: return doInlineClass();
37    default: return false;
38   }
39  }
40 }
```

Exemplo 7.1: Heurística para seleção e aplicação de *refactoring*.

A superclasse `RefactoringRandom` de `HillClimbing` tem a responsabilidade de configurar os parâmetros de um *refactoring* para ser aplicado ao sistema. O ponto de partida para execução da classe `HillClimbing` é o método `execute`.

A heurística do Exemplo 7.1 atua da seguinte forma:

1. é feito uma avaliação do sistema original (método `getAvaliation`) segundo a função de avaliação apresentada na seção 5.3;
2. é selecionado um *refactoring* de forma randômica e este é aplicado ao sistema (método `doRefactor`);
3. o software é novamente avaliado e comparado o valor da avaliação atual com o da última avaliação;
4. se a qualidade do sistema melhorou, a última avaliação (`lastAvaliation`) passa a ser a avaliação atual. Mas se a qualidade do sistema não melhorou, o último *refactoring* é desfeito (`undo`);
5. As etapas 2, 3 e 4 do ciclo se repetem por um número de iterações que é o critério de parada da heurística.

Quando um *refactoring* é selecionado é preciso prover as configurações adequadas para sua aplicação. Cada um dos *refactorings* tem sua própria estratégia de escolha aleatória das configurações, existindo 5 classe que implementam estas metodologias. São elas: `EncapsulateFieldRandom`, `MoveFieldRandom`, `MoveMethodRandom`, `ExtractClassRandom` e `InlineClassRandom`.

Ao ser selecionado o *Encapsulate Field*, a estratégia adotada é selecionar aleatoriamente um dos campos públicos do sistema. Este campo será encapsulado pelo *refactoring*. Não havendo nenhum campo público no sistema, a aplicação do *refactoring* é cancelada e um novo *refactoring* é selecionado de forma aleatória.

A estratégia para escolher o campo e as classes envolvidas na aplicação do *Move Field* é a seguinte. É selecionado aleatoriamente um campo qualquer do sistema, e por este campo

temos a classe origem do campo. A classe destino no campo também é selecionada de forma aleatória com a condição de estar no mesmo pacote da classe origem. Não havendo outra classe no mesmo pacote, um novo campo aleatório é selecionado. Se ocorrer de em 10 vezes o campo selecionado encontra-se em um pacote de única classe, a aplicação deste *refactoring* é cancelada.

O *refactoring Move Method* utiliza uma estratégia muito semelhante ao *Move Field*. Porém, o método é selecionado aleatoriamente entre todos os métodos não privados do sistema. A seleção da classe destino e o cancelamento do *refactoring* ocorrem na mesma situação descrita para o *Move Field*.

A estratégia para aplicação do *refactoring Extract Class* é mais bem elaborada que as três estratégias anteriores. Quando a heurística seleciona este *refactoring*, as classes do sistema são ordenadas pelo tamanho em relação ao número de membros. Três grupos de classes são criados então: classes pequenas, classes médias e classes grandes. A chance de se selecionar uma das classes pequenas é 10%, de se selecionar uma das médias é 30% e a chance que uma classe grande ser selecionada é 60% como ilustrado na Figura 7.2.

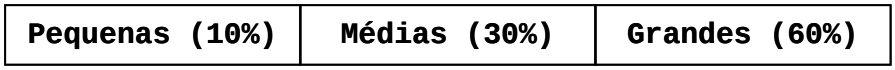

Figura 7.2: Chances das classes sofrerem *Extract Class*.

Uma vez selecionada a classe que irar sofrer extração de suas características (*Extract Class*). É preciso definir quais destas características (membros da classe) serão movidos para a nova classe e isto é feito escolhendo aleatoriamente alguns dos campos e métodos da classe. O número de membros movidos é sempre maior que 0 e menor que a quantidade de membros da classe. A classe que recebe as características é criada em um novo arquivo (no mesmo pacote da classe origem) e recebe o nome da classe origem adicionada do prefixo "`New`". Por exemplo, se a classe origem se chama `Aluno`, a nova classe irá se chamar `NewAluno` e será criada em um arquivo chamado `NewAluno.java`.

Para o *Inline Class* a estratégia de seleção das classes que serão unificadas ocorre da seguinte forma. Um par de classes que possui algum acoplamento entre elas é selecionado aleatoriamente no sistema. É considerado que há acoplamento entre duas classes quando ocorre uma das seguintes situações:

- uma das classes possui variável de instância que é do tipo da outra;
- uma das classes possui uma variável de classe (estática) que é do tipo da outra;
- é definida uma variável local a um método ou a um construtor em uma das classes que é do tipo da outra classe;
- um método ou construtor de uma classe recebe um argumento que é do tipo da outra classe;
- o tipo de retorno de um dos métodos de uma classe é o tipo da outra classe.

Um arquivo de propriedades chamado *System.txt* e que se encontra localizado no diretório *"...\guru\config"* deve ser configurado com as informações do sistema onde será feita a aplicação dos *refactorings*. As informações requeridas do sistema são o nome do sistema, o seu pacote padrão e o diretório raiz que possui todas as demais pastas e arquivos Java.

8 Trabalhos Relacionados

Existem alguns trabalhos em torno do tema de aplicação automática de *refactorings* e modelagem heurística para este problema. Três deles são citados neste capítulo e foram objetos de estudo para o desenvolvimento deste trabalho.

O primeiro destes trabalhos cujo título é "*Reformulating a Software Engineering as a Search Problem*" [Clarke et al, 1999] apresenta vários problemas típicos da engenharia de software que podem ser modelados como busca e aplicação de métodos heurísticos. Três destes problemas: "Teste do Sistema", "Modularização" (*Clustering*) e "Estimação de Custo" são discutidos e apresentadas modelagens utilizando metaheurísticas de busca local ou métodos genéticos. Neste mesmo artigo outros três problemas: "Fase de Coleta de Requisitos", "Integração do Sistema" e "Manutenção e Reengenharia usando Transformação de Programas" são lançados como possíveis candidatos à aplicação de métodos heurísticos sem um aprofundamento em sua modelagem. O último problema lançado em [Clarke et al, 1999] "Manutenção e Reengenharia usando Transformação de Programas" é onde atuamos buscando estudos aprofundados e uma proposta de modelagem.

Um segundo artigo que aborda o mesmo tema apenas dando enfoque um pouco diferente é "*Search-Based Software Engineering*" [Harman & Jones, 1999]. Este artigo argumenta sobre a modelagem de problemas da engenharia de software como problemas heurísticos de otimização. Nele são sugeridas estratégias para representação, construção da função de avaliação e operadores para aplicar a estrutura de representação.

Outro documento que possui certa relação com o que foi aqui apresentado é "*Envolving Object-Oriented Design with Refactorings*" [Tokuda, 1999] uma tese de doutorado que argumenta sobre a utilização de *refactorings* para adicionar padrões de projetos [Gamma et al, 1995] ao código dos programas. Por exemplo, são apresentados novos itens de *refactorings* e estes relacionados aos padrões *Bridge* e *Adapter*.

Estes trabalhos são apenas um exemplo do vasto número de opções que podem ser exploradas dentro deste tema de *refactoring* e aplicação automático.

9 Conclusões

O reprojeto dos sistemas é fundamental para sua evolução sendo a aplicação de refactprings um requisito essencial. Este projeto oferece condições para aplicação programada de *refactoring*, entretanto o maior objetivo é validar a aplicação automática.

A coleta de informações do sistema se mostra eficiente e este módulo pode ser utilizado de forma independente do restante da aplicação. Desta forma, JSystemInfo é um produto do projeto de grande aplicação prática não só para se obter métricas de software como também para identificar várias características presentes no código fonte.

A utilização de padrões de projeto [Gamma et al, 1995] é de suma importância para permitir a evolução de qualquer aplicativo. Nesta ferramenta, por exemplo, a aplicação do padrão *Visitor* permite que novas métricas sejam inclusas e o padrão *Singleton* foi utilizado para simplificar e dar clareza ao código.

A ferramenta para aplicação automática de *refactoring* nos oferece condições de avaliar sua utilização para aumentar a qualidade de softwares. É verificado ser possível a utilização de métodos heurísticos para seleção e aplicação dos itens de *refactoring*. Porém ainda não há garantia de que esta metodologia seja viável para qualquer tipo de sistema.

Quando a função de avaliação de qualidade de software, se pode constatar que esta ainda pode ser expandida e aperfeiçoada. Uma importe métrica que ainda não é absorvida pela função de avaliação é a coesão de uma classe. Esta deficiência pode provocar perda de coesão das classes da versão gerada do sistema objeto.

Como trabalhos futuros, é proposta a implementação de novos *refactorings*, a coleta de novas métricas de software e aperfeiçoamento da função de avaliação. É interessante também um trabalho para identificar e adicionar padrões de projetos através de informações de código e aplicação de *refactorings*.

Outra proposta futura é o acoplamento desta ferramenta a um Ambiente Integrado de Desenvolvimento de Sistemas (IDE) para facilitar sua utilização por parte da comunidade de desenvolvedores.

Referências Bibliográficas

[Beck, 1999] Beck, Kent. *Extreme Programming Explained – Embrace Change*. Addison Wesley, 1999.

[Brand & Visser, 1980] Brand, Mark Van Den e Visser, Eelco. *Generation of Formatters for Context-free Languages*. In ACM Transactions on Software Engineering and Methodology, v.5, n. 1, pp. 1-41. 1996.

[Castor & Borba, 2001] Castor, Fernando e Borba, Paulo *A Language For Specifying Java transformations*. In V Simpósio Brasileiro de Linguagens de Programação, pp.236-251. 2001. Curitiba, Brazil.

[Castor et al., 2001] Castor, F., Oliveira, K., Souza, A., Santos, G., e Borba, Paulo. *JaTS: A Java Transformation System*. In XV Simpósio Brasileiro de Engenharia de Software, pp. 374-379. 2001. Rio de Janeiro, Brazil.

[Chidamber & Kemerer, 1994] Chidamber, S. e Kemerer, C. *A Metrics Suite for Object Oriented Design*. In IEEE Transaction on Software Engineering, Volume 20, N. 6 pp. 476-493, June 1994.

[Clarke et al, 1999] Clarke, J., Dolado J. J., Harman M., Jones B., Lumkin M., Mitchell B., Rees K., Roper M., e Shepperd, M. *Reformulating Software Engineering as a Search Problem*. By SPSRC SEMINAL, 1999.

[Donaldson et al., 1981] Donaldson, J., Lancaster, A. e Sposato, P. *A Plagiarism Detection System*. In SIGCSE Technical Symposium on Computer Science Education, pp. 21-25, 1981.

[Fowler, 1999] Fowler, Martin. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R. e Vlissides, J. *DESIGN PATTERNS – Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[Harman & Jones, 1999] Harman, Mark e Jones, Bryan F. *Search-Based Software Engineering*. Disponível na URL: http://www.brunel.ac.uk/~csstmmh2/papers.html em 10 de Novembro de 2003. United Kingdom, 1999.

[Harrison et al, 1998] Harrison, R., Counsell S. J. e Nithi, R. V. *An Evaluation of the MOOD Set of Object-Oriented Software Metrics*. Department of Electronics and Computer Science, University of Southampton. UK, 1998.

[Kan, 2003] Kan, Stephen H. *Metrics and Models in Software Quality Engineering*, 2nd ed. Pearson Education, 2003.

[Kerievsky, 2002] Kerievsky, Joshua. *Refactoring To Patterns*, v.0.17 (em desenvolvimento). Industrial Logic Inc, **2002**. [online] Disponível na Internet via WWW. URL: http://www.industriallogic.com/xp/refactoring/ em 07 de Julho de 2003.

[Kirkpatrick et al, 1983] Kirkpatrick, S., Gellat, D. C. e Vecchi, M. P. *Optimization by Simulated Annealing*. Science Magazine, p. 671–680, 1983.

[Lorenz & Kidd, 1994] Lorenz, M. e Kidd J. *Object-Oriented Software Metrics, A Practical Guide*. Englewood Cliffs, N.J.: PTR Prentice-Hall, 1994.

[Maia & Oliviera, 2002] Maia, Marcelo A. e Oliveira, Ademir A. *JPearl - Uma linguagem para descrever reestruturações de programas Java.* In VI Simpósio Brasileiro de Linguagens de Programação. Rio de Janeiro, 2002.

[Oliveira, 2002] Oliveira, Ademir A. *JMJ - Uma linguagem de domínio específico para manipulação de programas*. Ouro Preto, 2002. Trabalho final de graduação em Ciência da Computação – DECOM, Universidade Federal de Ouro Preto.

[Opdyke, 1992] Opdyke, William F. *Refactoring Object-Oriented Frameworks* . University of Illinois at Urbana-Champaign, 1992. Ph. D Thesis.

[Oppen, 1980] Oppen, Derek C. *Prettyprinting*. In ACM Transactions On Programming Language and Systems. v.2, n.4, pp. 465-483, 1980.

[Partsch & Steinbrüggen, 1983] Partsch, H. E Steinbrüggen, R. *Program Transformation Systems*, ACM Computing Surveys, v.15, n.3, pp. 199-236, 1983.

[Pressman, 1995] Pressman Roger S. *Engenharia de Software*. 3ª edição, Makron Books, 1995.

[Roberts, 1999] Roberts, Donald Bradley. *Practical Analysis for Refactoring*. Department of Computer Science. University of Illinois at Urbana-Champaign, 1999.

[Roberts et al., 1997] Roberts, Don, Brant, John e Johnson, Ralph *A Refactoring Tool for Smaltalk*. University of Illinois at Urbana-Champaign, 1997.

[Souza, 2002] Souza, Marcone J. F. *Inteligência Computacional para Otimização*. Notas de Aula. Departamento de Computação, UFOP, 2002.

[Tokuda, 1999] Tokuda, Lance Aiji. *Envolving Object-Oriented Design with Refactorings*. Ph. D Thesis. University of Texas at Austin, 1999.

[Wise, 1992] Wise, Michael J. *Detection of similarities in student programs: YAP'ing may be preferable to Plague'ing*. In ACM SIGSCI, pp. 268-271, 1992. Kansas City, USA

Bibliografia Consultada

Fowler, Martin. *"Refactoring Home Page"*. [online] Disponível na internet via WWW. URL:http://www.refactoring.com. Consultado em 26 de junho de 2003.

Instantiations, Inc. *"JFactor Home Page"*. [online] Disponível na internet via WWW. URL: http://www.instantiations.com/jfactor/. Consultado em 10 de julho de 2003.

JRefactory. *"JRefactory Home Page"*. [online] Disponível na internet via WWW. URL: http://jrefactory.sourceforge.net/. Consultado em 10 de julho de 2003.

RefactorIt. *"RefactorIt – Java Refactoring Tool"*. [online] Disponível na internet via WWW. URL: http://www.refactorit.com/. Consultado em 10 de julho de 2003.

Wells, Don *"Exteming Programming"*. [online] Disponível na internet via WWW. URL: http://www.extremeprogramming.org. Consultado em 30 de junho de 2003.

Anexo A – *Refactoring Encapsulate Field*

```
1  package jmj.refactorings.fowler.encapsulateField;
2  public composed metaCode EncapsulateField {
3   String inputFile;  // Arquivo de entrada
4   String outputFile; // Arquivo de saida
5   String className;
6   String fieldName;

7   public EncapsulateField(String ifl, String ofl, String cn, String fn) {
8    inputFile = ifl;
9    outputFile = ofl;
10   className = cn;
11   fieldName = fn;
12  }

13  public void execute() throws ActionException {
14   if (condition()) {
15    MetaCode matchRule = #new ClassWithField(inputFile);
16    MetaCode outputRule = #new FieldWithGetAndSetMethods(outputFile);
17    matchRule.setVariable( "className", getIdentifier( className ) );
18    matchRule.setVariable( "fieldName", getIdentifier( fieldName ) );
19    if (matchRule.executeMetaCode () == SUCCESS) {
20     bind(matchRule, outputRule);
21     if ( !( (matchRule.#fm).isMatched() ) )
22      outputRule.setVariable( "fm2", getFieldModifiers( "private" ) );
23     else {
24      #FieldModifiers fm2 = ((#FieldModifiers)matchRule.#fm);
25      fm2.remove ("public");
26      fm2.remove ("protected");
27      if ( !(fm2.isPrivate()) ) fm2.add ("private");
28      outputRule.setVariable( "fm2", fm2 );
29     }
30     outputRule.setVariable( "setMethod", getIdentifier(getSetMethod()));
31     outputRule.setVariable( "getMethod", getIdentifier(getGetMethod()));
32     if (outputRule.executeMetaCode () != SUCCESS)
33      throw new ActionException("Cannot save "+outputFile+".");
34    } else throw new ActionException(fieldname + " was not found.");
35   } else throw new ActionException("Get or set method declared.");
36  }
             // Verifica se nao existem metodos get e set em className
37  public boolean condition() {
38   MetaCode rlGetVerify = #new ClassWithMethod(inputFile);
39   rlGetVerify.setVariable ("className", getIdentifier(className) );
40   rlGetVerify.setVariable ("methodName", getIdentifier(getGetMethod()));
41   MetaCode rlSetVerify = #new ClassWithMethod(inputFile);
42   rlSetVerify.setVariable ("className" , getIdentifier(className) );
43   rlSetVerify.setVariable("methodName", getIdentifier(getSetMethod()));
44   return (rlGetVerify.executeMetaCode()!=SUCCESS) &&
   (rlSetVerify.executeMetaCode()!=SUCCESS);
45  }

46  public String getSetMethod() { return ("set" +
   fieldName.substring(0,1).toUpperCase() + fieldName.substring(1));
47  }
48  public String getGetMethod() { return ("get" +
   fieldName.substring(0,1).toUpperCase() + fieldName.substring(1));
49  }
50 }
```

Anexo B – *Refactoring Move Method*

```
1  package jmj.refactorings.fowler.moveMethod;

2  import jmj.refactorings.fowler.util.add.method.*;

3  public composed metaCode MoveMethod {
4   String inputSourceFile;  // Arquivo que contem a classe origem
5   String outputSourceFile; // Arquivo com a classe origem reestruturado
6   String inputTargetFile;  // Arquivo que contem a classe destino
7   String outputTargetFile; // Arquivo gerado que contem a classe destino
8   String sourceClass;      // Nome da classe origem
9   String targetClass;      // Nome da classe destino
10  String methodName;       // Nome do metodo a ser movido

11  public MoveMethod(String isf, String osf, String itf, String otf,
   String sc, String tc, String mn) {
12   inputSourceFile = isf;
13   outputSourceFile = osf;
14   inputTargetFile = itf;
15   outputTargetFile = otf;
16   sourceClass = sc;
17   targetClass = tc;
18   methodName = mn;
19  }

20  public void execute() throws ActionException {
        // VerifyConcreteMethod
21   MetaCode rlVerifyCcMt = #new ClassWithConcreteMethod(inputSourceFile);
22   rlVerifyCcMt.setVariable ("className", getIdentifier(sourceClass));
23   rlVerifyCcMt .setVariable ("methodName", getIdentifier(methodName));
24   if (rlVerifyCcMt .executeMetaCode() == SUCCESS) {
        // AddMethod
25    AddMethod mcAddMethod = new AddMethod(inputTargetFile,
   outputTargetFile, targetClass, rlVerifyCcMt.#mm.toString(),
   rlVerifyCcMt.#t.toString(),methodName, rlVerifyCcMt.#fpms.toString(),
   rlVerifyCcMt.#tc.toString(), rlVerifyCcMt.#blck.toString());
26    mcAddMethod.execute();
        // RedoMethod
27    RedoMethod mcRedoMethod = new RedoMethod(outputTargetFile,
   outputTargetFile, targetClass, methodName, sourceClass);
28    mcRedoMethod.execute();
        // RedirectTarget
29    RedirectTarget mcRedirectTarget = new RedirectTarget(
   outputTargetFile, outputTargetFile, targetClass, methodName,
   mcRedoMethod.getFieldName());
30    mcRedirectTarget.execute();
        // RedirectSource
31    RedirectSource mcRedirectSource = new RedirectSource(
   inputSourceFile, outputSourceFile, sourceClass, methodName,
   targetClass);
32    mcRedirectSource.execute();
33   } else throw new ActionException("Move Method: Condition FAIL");
34  }
35 }
```

Anexo C – *Refactoring Extract Class*

```
1  package jmj.refactorings.fowler.extractClass;

2  import jmj.refactorings.fowler.util.add.addClass.*;
3  import jmj.refactorings.fowler.util.add.fieldDeclaration.*;
4  import jmj.refactorings.fowler.moveField.*;
5  import jmj.refactorings.fowler.moveMethod.*;

6  public composed metaCode ExtractClass {
7   String inputSourceFile;   // Arquivo que contem a classe com o campo
8   String outputSourceFile;  // Arquivo que contem a classe reestruturado
9   String inputTargetFile;   // Arquivo para onde vai a nova classe
10  String outputTargetFile; // Arquivo com a  nova classe gerada
11  String sourceClass;      // Classe original
12  String targetClass;      // Nome da nova classe a ser gerada
13  Vector fields = new Vector();  // Campos que irao para a nova classe
14  Vector methods = new Vector(); // Metodos que irao para a nova classe

15  public ExtractClass(String isf, String osf, String itf, String otf,
   String sc, String tc) {
16   inputSourceFile = isf;
17   outputSourceFile = osf;
18   inputTargetFile = itf;
19   outputTargetFile = otf;
20   sourceClass = sc;
21   targetClass = tc;
22  }

23  public void execute() throws ActionException {
        // CreateClass
24   CreateClass mcCreateClass = new CreateClass(inputTargetFile,
   outputTargetFile, "public", targetClass);
25   mcCreateClass.execute();
        // AddField
26   AddField mcAddField = new AddField(inputSourceFile, outputSourceFile,
   sourceClass, targetClass, "my"+targetClass, "=new "+targetClass+"()");
27   mcAddField.execute();
        // MoveField
28   MoveField mcMoveField = new MoveField(outputSourceFile,
   outputSourceFile, outputTargetFile, outputTargetFile, sourceClass,
   targetClass );
29   while ( !(fields.isEmpty()) ) {
30    mcMoveField.setFieldName((String)fields.remove(0));
31    mcMoveField.execute();
32   }
        // MoveMethod
33   MoveMethod mcMoveMethod = new MoveMethod(outputSourceFile,
   outputSourceFile, outputTargetFile, outputTargetFile, sourceClass,
   targetClass);
34   while ( !(methods.isEmpty()) ) {
35    mcMoveMethod.setMethodName((String)methods.remove(0));
36    mcMoveMethod.execute();
37   }
38  }

39  public void addField(String field) { fields.add(field); }
40  public void addMethod(String method) { methods.add(method); }
41 }
```

Anexo D – *Refactoring Inline Class*

```
1  package jmj.refactorings.fowler.inlineClass;

2  import java.io.File;
3  import templates.java.units.NoneType;
4  import jmj.refactorings.fowler.inlineClass.aid.*;
5  import jmj.refactorings.fowler.moveField.*;
6  import jmj.refactorings.fowler.moveMethod.*;
7  import jmj.refactorings.fowler.util.remove.fieldDeclaration.RemoveField;
8  import jmj.refactorings.fowler.util.remove.delegatingMethod.*;
9  import jmj.refactorings.fowler.util.remove.removeClass.RemoveClass;

10 public composed metaCode InlineClass {
11  String inSrcFile; // Arquivo que contem a classe absorvida
12  String outSrcFile; // Arquivo que sem a classe (removida)
13  String inTgtFile; // Arquivo para onde vai as caracteristicas
14  String outTgtFile; // Arquivo com as novas caracteristicas
15  String sourceClass;
16  String targetClass;
17  private Vector _fields = new Vector();
18  private Vector _methods = new Vector();

19  public InlineClass(String isf, String osf, String itf, String otf,
   String sc, String tc) {
20   inSrcFile = isf;
21   outSrcFile = osf;
22   inTgtFile = itf;
23   outTgtFile = otf;
24   sourceClass = sc;
25   targetClass = tc;
26  }

27  public void execute() throws ActionException {
28   Features features = new Features(inSrcFile, sourceClass);
29   features.execute();
30   _fields = features.getFields();
31   _methods = features.getMethods();
32   int fdIndex = 0;
33   int mtIndex = 0;
              // MoveField
34   while ( !(_fields.isEmpty()) ) {
35    MoveField moveField = new MoveField(inSrcFile, outSrcFile, inTgtFile,
   outTgtFile, sourceClass, targetClass);
36    String field = (String)_fields.get(fdIndex++);
37    moveField.setFieldName(field);
38    moveField.execute();
39   }
40 // MoveMethod
41   while ( !(_methods.isEmpty()) ) {
42    MoveMethod moveMethod = new MoveMethod( inSrcFile, outSrcFile,
   inTgtFile, outTgtFile, sourceClass, targetClass );
43    String method = (String)_methods.get(mtIndex++);
44    moveMethod.setMethodName(method);
45    moveMethod.execute();
46   }
              // Remove souce reference on target class
47   RemoveField delField = new RemoveField(outTgtFile, outTgtFile,
   targetClass);
```

```
48   delField.setType( sourceClass );
49   delField.execute();
               // Find source refererences on package classes
50   File inputFile = new File(inSrcFile);
51   String mainDir = inputFile.getParent()+"/";
52   File dir = new File(mainDir);
53   String[] jFiles = getJavaFiles(dir.list());
54   for (int i=0; i<jFiles.length; i++) {
55    String javaFile = mainDir+jFiles[i];
56    JavaFileClass fileClasses = new JavaFileClass(javaFile);
57    fileClasses.execute();
58    Vector classes = fileClasses.getClassNames();
59    while ( !(classes.isEmpty()) ) {
60     String classe = (String)classes.remove(0);
61     RemoveField delField = new RemoveField(javaFile, javaFile, classe);
62     delField.setType( sourceClass );
63     delField.execute();
64    }
65   }
               // Remove souce file or source class
66   RemoveClass remove = new RemoveClass(inSrcFile, inSrcFile,
   sourceClass);
67   remove.execute();
68   NoneType tnotype = new NoneType();
69   if (tnotype.match(inSrcFile)) inputFile.delete();
70   File classFile = new File(inSrcFile.replaceFirst(".java", ".class"));
71   classFile.delete();
72  }

73  String[] getJavaFiles(String files[]) {
74   int j = 0;
75   for(int i=0; i<files.length; i++) if(files[i].endsWith(".java")) j++;
76   String javaFiles[] = new String[j];
77   j = 0;
78   for(int i=0; i<files.length; i++)
79    if(files[i].endsWith(".java")) javaFiles[j++] = files[i];
80   return javaFiles;
81  }
82 }
```

Anexo E – Classe *Refactor* (Aplicadora de *refactorings*)

```
1  package guru.heuristic;

2  import java.io.File;
3  import javax.swing.*;
4  import guru.Config;
5  import guru.out.*;
6  import guru.undo.*;

7  public class Refactor {
8   public static final String TEMPDIR = Config.PROJDIRETORY+"/guru/temp/";
9   public static File undoFilesDir = new File(UndoManager.UNDOFILESDIR);
10  private int stateNumber = 0;
11  private String callfile = TEMPDIR+"jmj.call";
12  FileWriter writer;
13  UndoManager undo = new UndoManager();

14  public Refactor() {
15   File[] files = undoFilesDir.listFiles();
16   for (int i=0; i<files.length; i++) files[i].delete();
17  }

18  public void appEncapsulateField(String fn, String cn, String fd) {
19   State state = new State(stateNumber++);
20   state.refactor(fn);
21   undo.pushState(state);
22   writer = new FileWriter(TEMPDIR+"jmj.call");
23   writer.writeEncapsulateFieldFile(fn, fn, cn, fd);
24   execute();
25  }

26  public void appMoveField(String sf, String tf, String sc, String fd,
   String tc) {
27   State state = new State(stateNumber++);
28   state.refactor(sf);
29   state.refactor(tf);
30   undo.pushState(state);
31   writer = new FileWriter(TEMPDIR+"jmj.call");
32   writer.writeMoveFieldFile(sf, sf, tf, tf, sc, fd, tc);
33   execute();
34  }

35  public void appMoveMethod(String sf, String tf, String sc, String mt,
   String tc) {
36   State state = new State(stateNumber++);
37   state.refactor(sf);
38   state.refactor(tf);
39   undo.pushState(state);
40   writer = new FileWriter(TEMPDIR+"jmj.call");
41   writer.writeMoveMethodFile(sf, sf, tf, tf, sc, mt, tc);
42   execute();
43  }

44  public void appExtractClass(String sf, String tf, String sc, String tc,
   String[] fds, String[] mts) {
45   State state = new State(stateNumber++);
46   state.refactor(sf);
47   state.create(tf);
```

```
48   undo.pushState(state);
49   writer = new FileWriter(TEMPDIR+"jmj.call");
50   writer.writeExtractClassFile(sf, sf, tf, tf, sc, tc, fds, mts);
51   execute();
52  }

53  public void appInlineClass(String sf, String tf, String sc, String tc){
54   State state = new State(stateNumber++);
55   state.refactor(sf);
56   state.refactor(tf);
57   undo.pushState(state);
58   writer = new FileWriter(TEMPDIR+"jmj.call");
59   writer.writeInlineClassFile(sf, sf, tf, tf, sc, tc);
60   execute();
61  }

62  protected void execute() {
63   JMJCaller caller = new JMJCaller(callfile);
64   caller.execute(true);
65  }

66  public void undo() {
67   undo.undo();
68  }
69 }
```

UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO - DECOM
CAMPUS UNIVERSITÁRIO - MORRO DO CRUZEIRO
CEP 35400-000 - OURO PRETO - MINAS GERAIS